POLITECNICO DI TORINO

SCUOLA DI DOTTORATO

Course of Information and System Engineering (ING–INF/05) – XXV cycle

Doctoral thesis

High Speed and Flexible Network Processing

Marco Leogrande

Tutor
Fulvio Risso

Course Coordinator
prof. Pietro Laface

March 21st, 2014

Summary

In digital communications networks, **packet processing** refers to the wide variety of algorithms that are applied to a packet of data or information as it moves through the various network elements of a communications network. Within any network-enabled device, it is the packet processing subsystem that manages the traversal of the multi-layered network or **protocol stack** from the lower, physical and network layers all the way through to the application layer.

There are a number of important requirements that a packet processing system must observe to be able to meet real-world demand.

- The first and probably more obvious requirement is to achieve an high **processing speed**. To be able to cope with the amount of information exchanged over a present-day computer network, a packet processing system should be able to perform its operations at line rate. In other words, its architecture and algorithms should be designed with **efficiency** in mind, so that most of the traffic (if not all) flows through the *fast path* of the system, whereas only special cases are handled by a more thorough, albeit slower logic[1].

- Network devices should also be **flexible**, to adapt to the user's environment and needs. Given the complexity and variety of contemporary computer networks, packet processing systems should offer a high degree of configurability; no solution can encompass all possible network sizes, topologies and implementations, but being able to adapt to different layouts is a key aspect for any network device.

- Traditionally, network devices have been closed-source, black boxes with little or no possibility of changing or replacing modules. Even if it has always been possible to put together multiple devices to form a complex system, each device has always been considered an inscrutable unit. However, especially after that the paradigm of *Software Defined Networks* has proven that it is possible to efficiently insert third-party modules inside a running network, the topic of **modularity** has risen in importance. Having a packet processing system with a plug-in architecture, or whose external dependencies are easy to replace, is a very strong selling point.

[1]For instance, network devices usually implement the *data plane* with fast and specifically-designed hardware, while the *control plane* is run by software running on the control processor.

- Given the quantity and unpredictability of the events that can occur on a computer network, a packet processing system should also be **dynamic** enough to be able to respond and adapt to real-time fluctuations in the network. Also, consider that the network administrator might want to interact with the network in real-time, adding further potential instability to the system: a packet processing system should be able to cope with all those events happening at the same time.

Packet filters

Packet filters are a specific type of packet processing systems. Packet filtering is the process of inspecting a network packet and matching it against a set of predefined rules. Rules usually have associated actions, that are executed when a given rule matches, and are usually dependent on the specific system in which a packet filter resides.

For instance, a *network firewall* uses a packet filter to match incoming and outgoing packets against the firewall rules, with the goal of applying the policy that the matching rule has defined (e.g., drop the packet, or send it out of a given interface, or apply address translation, etc. . .).

Moreover, a packet filter is said to be **dynamic** when the decision to apply a rule is not only based on the content of the packet under investigation (that would be a **static** packet filter), but also on additional metadata or context information. For instance, a device that performs Network Address Translation (NAT) has to keep the list of active connections, to be able to successfully differentiate packets that are part of an open session from those that are not, with the possible intent of opening a new session or rejecting it.

Packet filters are used in a variety of systems; a few of them have been listed below.

- A **firewall** is used to help keep a network secure. Its primary objective is to control the incoming and outgoing network traffic by analyzing the data packets with a packet filter and determining whether it should be allowed through or not. A network's firewall builds a bridge between an internal network that is assumed to be secure and trusted, and another network, usually an external network such as the Internet, that is not assumed to be secure and trusted.

- An **Intrusion Detection System** (IDS) is used to monitor network packets or systems for malicious activity and perform a specific action if such activity is detected. Usually, if malicious activity is detected on the network, the source IP of the malicious traffic is blocked for a certain period of time, and all of the packets from that IP address will be rejected.

- An **Intrusion Prevention System** (IPS) is basically an upgrade of an intrusion detection system. Where the IDS is used to detect, block and log the attack, the IPS is used also to prevent the attack itself, either in advance or while it is happening.

- A system that performs **Security Information and Event Management** (SIEM) can monitor security alerts generated by various software or hardware solutions that are used for detecting malicious activity. SIEM itself is a combination of:

– SIM (Security Information Management): provides the analysis and reporting of the logged data.
– SEM (Security Event Management): provides monitoring and correlation of events.

In this context, a packet filter can be used as one of the sources of events to be aggregated and analyzed.

Present and future challenges for packet filters

Packet filter technologies are facing new issues every day, as we had to re-engineer our computer networks in order to accommodate many new use cases. For instance, low-level network protocols are growing in number: new solutions, arising in particular for the purpose of network virtualization (e.g., 802QinQ, VXLAN), are rapidly transforming the Ethernet frames. The middle layers of the protocol stack are facing a similar metamorphosis: examples include the widespread adoption of Virtual Private Networks, or the necessity to transport IPv6 traffic over IPv4 networks.

Packet filters are dramatically affected by those changes, as they become more complicated: it is important to be able to capture all the traffic we are interested in (e.g., web traffic), independently from the actual encapsulation used at lower layers.

For this reason, the scientific research should embrace these new issues by proposing improvements over the traditional technologies, with the goal of maintaining the standards of efficiency of flexibility that we are used to.

This dissertation addresses two specific issues:

1. How to preserve packet filter **flexibility** when specifying packet matching rules. We need a solution that allows a finer specification of matching rules, but that is also independent (if desired) on the specific encapsulation used at lower levels; moreover, the solution should support protocol definitions specified at run-time. Part I addresses the problem and describes in detail the proposed solution: **NetPFL**, a declarative language targeted to data-plane packet processing.

2. How to achieve **efficiency** when representing and combining multiple packet filters, even in case of bizarre and unusual network encapsulations. Part II outlines the issue and proposes two solutions: **pFSA** (described in Chapter 2) and its extension, **xpFSA** (delineated in Chapter 3).

Contents

List of Tables

List of Figures

Part I

Solving the expressivity problem for packet filter languages

Chapter 1

NetPFL: a declarative language for data-plane packet processing

1.1 The issues with current packet filter languages

Many networking tools based on protocol parsing (packet filters, firewalls, Intrusion Detection Systems, etc.) were developed in a time when the number of protocols was limited and the encapsulation relationships among them were rather simple.

Currently this assumption has become no longer valid. The number of possible protocol encapsulations in network traffic is growing day after day due to several reasons: the attempt to bypass application-layer constraints and, in general, to escape network restrictions (e.g., different application protocols transported in HTTP in order to bypass firewalls), or the necessity to establish Virtual Private Network sessions in many different environments and to transport unsupported traffic. Many of these solutions require the encapsulation of lower-layer protocols in other ones of the same level (as IPv6 in IPv4), or even in higher-layer ones (e.g. IP in UDP).

Although current packet filtering tools are able to support common conditional predicates (e.g, the presence of a protocol or a field with a given value), they fail when requested to select only specific types of encapsulation (e.g. IPv6 traffic encapsulated in IPv4). This is a problem for the aforementioned network-based tools, whose solution requires the addition of a module that is able to selectively inspect the network traffic before the actual processing; this strategy is, however, error-prone and adds computational overhead. Furthermore, the list of supported encapsulations is generally hardcoded in the packet filtering tools (therefore not easily expandable) and it is often limited with respect to tunneled protocols. This represents an additional limitation because the packet filter needs to be recompiled in order to extend its supported protocol set.

To overcome the previous limitations in filtering expressivity, a new packet filtering language is needed. This Part introduces **NetPFL** (abbreviation for *Network Packet Filtering Language*), a new declarative language for data-plane packet processing. Among its main strengths, *(i)* it can handle complex situations of tunneled and stacked encapsulations, giving the user a finer tuned control over the semantics of a filtering expression; *(ii)*

it supports several independent filters that can lead to multiple matches; *(iii)* it allows the user to choose, in an implementation-agnostic way, the action that should be performed upon receipt of each matching packet; *(iv)* associates each packet with a stream indicator, to facilitate the merging of different logical filters within the same physical filtering machine and the demultiplexing of the associated packets that belong to different logical filters; *(v)* it is human-friendly, making it suitable for fast command-line processing.

Furthermore, no protocol knowledge is embedded in NetPFL, facilitating its integration with any other language that provides protocol specification, e.g. external protocol databases that can be defined (and updated) independently from the packet filtering language. Particularly, our prototypical implementation associates NetPFL to the NetPDL [28] language, which provides the protocol definition, enabling support for run-time updates of the protocol data set.

The state of the art for packet filter languages is presented in Section 1.2, while the proposed language specification for NetPFL is discussed in Section 1.3. Section 1.12 draws the conclusions about NetPFL.

1.2 State of the art in packet filter languages

Current packet filtering solutions are based on different filtering languages that look extremely similar, as they are engineered with similar purposes in mind, but are often different with respect to the language details or the approach used to express Boolean conditions. A non-exhaustive listing of the most common packet filtering languages is depicted here.

libpcap [8] is probably the most famous packet capture (and filtering) library; it runs on top of many UNIX-based kernels (and Windows, in its WinPcap flavor [1]) and exposes to the filtering applications a set of primitives [2] that can be combined to fully express the desired capture syntax. Predicates operate on selected fields of some of the most common protocols (i.e. Ethernet, IEEE 802.11, IP, TCP, UDP and others) or on the length of the packet. **Wireshark** [3] is a popular packet sniffing and analysis application that, while relying on libpcap as primary packet filtering engine, uses its own syntax in post-processing mode. This language allows a broader set of filters to be expressed, both in terms of allowed predicates and in terms of protocol and fields supported; the official website states that, as of version 1.2.6, over 85000 protocol fields can be specified. Both libpcap and Wireshark, as many others, allow to define a filter that matches a selected number of protocols (e.g., `l2tp`), but fail when requested to express different conditions for the encapsulating protocols (e.g., `l2tp in ipv6`) and do not support matching against multiple filters. Furthermore, they do not have action capabilities and have static protocol descriptions hardwired in their code, making extensibility cumbersome.

The last problem is solved by the **NetPDL** [28] language, which describes protocol formats and encapsulation rules. However, this language is purely descriptive and it does not specify any primitive to define the actual filter, requiring another language for defining the filtering expression based on its extensible protocol description. **Binpac** [4] is similar to NetPDL since it focuses on protocol description, although its many primitives also enable the definition of generic actions. However, the language focuses on application-layer

protocols, and it requires full programming, making it not suitable for fast, command-line interface commands.

While packet filtering is a fundamental part of most networking applications, often the resulting packet stream has to be further processed in order to complete the application's job. These applications often define their own language, including both filtering primitives and a specification of the actions to be applied to the resulting packet stream. Some well-known examples can be found in some popular Intrusion Detection Systems (IDS) such as **Snort** [5] or **Bro** [6], which allow only a handful of protocol fields to be inspected, focusing instead on the action to be performed or on the payload of the transport protocol. Still, primitives required to differentiate tunneling do not exist and the protocol set is hardcoded in the application, but in this case they are able to define some complex actions, such as inspect the payload, raise an alert, and more. In fact, their languages focus on the peculiar set of actions required by the application, with limited possibility to reuse (or extend) that language in case of a different application. Differently from the previous applications, IDSs are able to define multiple rules (i.e. filters) that can be active at the same time and that can lead to multiple matching; the receiving module is made aware of the filters that matched against each packet.

Similarly, also **Stream-SQL** [7] focuses on high-level actions, enabling sophisticated elaborations (e.g. grouping, counting, ordering, etc.) on a data live stream through a SQL-like syntax. However, this language does not include the traditional filtering capabilities operating on packets. Filtering primitives (i.e., the `SELECT` keyword) operate on a live stream that looks like a structured table, as in a traditional database, making this approach unfeasible for classical packet filtering. For the same reason, the number of protocols and fields identified is limited to those known by the engine that pre-processes the network traffic and creates the live stream in tabular format.

All the approaches presented above suffer of at least one of the following problems:

- **no tunneling support**: even if tunneling protocols are successfully recognized, multiple instances of the same protocol in the same packet cannot be treated separately; furthermore, the user cannot select precisely which encapsulations have to be considered when capturing packets;
- **no multiple independent filters**: most of the languages do support multiple independent (and potentially overlapping) filters that can be satisfied at the same time, and do not allow to return the list of the matching filters to the application;
- **limited actions and no extensibility**: each language aims at solving only the problem of the particular application and there is no provision of a generic action-based language that can support many applications;
- **human-friendliness**: we want the language to be used on a command line tool, in order to be able to quickly react to changing network conditions without having to rely on complex building toolchains;
- **hardwired protocol description**: the number of protocols recognized by each implementation can be increased only by editing the source code of the application

and recompiling it.

NetPFL aims at solving the first four problems: appropriate (and human-friendly) primitives enable the identification and selection of each protocol in a tunnel, while keeping an high degree of flexibility. Furthermore some common actions have been defined, which can be further extended at will, and multiple independent filters are supported. With respect to the fifth problem, NetPFL is completely protocol-agnostic, as it has no *a priori* knowledge about the structure of the packets to be processed. NetPFL relies on other languages to describe protocol formats: in our implementation it has been associated with NetPDL.

1.3 Introduction to NetPFL

The Network Packet Filtering Language (*NetPFL*) is a declarative language targeted to data-plane packet processing. It implements a *filter - action* paradigm for processing network packets, which consists in creating a filtering expression and to define a possible set of actions to be executed in case the filtering condition is satisfied.

While the syntax of NetPFL allows defining a complex filtering rule based on *protocols* and *fields*, it does not define the list of protocols/fields supported. For instance, the names of protocols and fields are not defined as special keywords of the language, but are dynamically bound to those defined in an external data set. The *protocol independence* of NetPFL enables this language to be used for any present and future protocol.

NetPFL can be viewed as complementary to NetPDL [28], because it directly exploits the capability of NetPDL to describe the binary format of network protocol headers and encapsulation relationships between different protocols. In particular, while NetPDL describes the content of each packet in terms of headers and fields, NetPFL defines packet filtering expressions, as well as associated actions that should be performed on a packet when the above-mentioned filtering conditions are verified.

The basic syntax of the language is introduced first in Section 1.4, then each subsequent section analyzes all language constructs in detail. Starting from Section 1.9 we describe the advanced features of NetPFL for dealing with uncommon forms of encapsulation, such as those involving tunneling.

1.4 NetPFL general syntax

NetPFL is a rule-based language following a filter-action model to express packet handling statements, such as accepting a packet, or extracting the actual values of a set of fields. The basic syntax for a NetPFL rule is the following:

```
[FilteringExpression] [Action] [as stream <StreamID>]
```

The rule is applied to every incoming packet with the following semantic:

when `FilteringExpression` *is true, perform* `Action` *and associate the current packet and*

the results of the action to the stream identified by `StreamID`

The filtering expression is a Boolean function that can be based on (*i*) protocols (i.e. a filter is satisfied if the packet contains the specified protocol header), and (*ii*) checks on field values (i.e. a filter can be specified as an expression involving the value of one or more header fields). Basic predicates can be composed with the Boolean operators AND, OR, and NOT in order to express complex filters. Since the filtering expression is an optional part of a NetPFL statement, when a filter is not specified, the action is applied to all incoming packets. The detailed syntax of filtering expressions is described in Section 1.6.

Currently NetPFL defines two actions: (*i*) `returnpacket`, for simply accepting packets satisfying `FilteringExpression`, and (*ii*) `extractfields` for extracting the values of a list of fields specified by the user. `returnpacket` is the default action. Actions are described in Section 1.7.

In NetPFL, both the set of all the packets accepted by a filter and the set of all the tuples of fields extracted from consecutive packets are called "streams". The user can specify an identifier for each NetPFL statement through the optional trailing construct "`as stream <StreamID>`", where `<StreamID>` is a numeric identifier. When multiple rules are defined, this allows the user to recognize which rule(s) matched a packet. In particular, when more than one rule has a match, the set of the stream IDs corresponding to such rules is returned to the user. The actual format of this set is implementation-dependent. Streams are discussed in Section 1.8.

1.5 Filtering Elements

1.5.1 Protocol and Field Identifiers

In NetPFL, protocol and field identifiers are not tokens specified in the language, but are derived from the associated protocol database. In other words, the resolution of the names of protocol and fields used in a NetPFL rule is transparent to the language, which only requires that such identifiers correspond to valid protocol and field names within the database in use.

Protocol Identifiers

A *protocol identifier* is simply a string literal corresponding to a protocol defined in the NetPDL database. The resolution of the protocol name is case-insensitive. An example of protocol identifier is given by the string "`ip`".

Field Identifiers

A *field identifier* is represented as a protocol identifier followed by a "`.`" (dot), followed by the name of a valid field for the specified protocol, e.g.

proto-name . *field-name*

The resolution of a field name is case-insensitive. An example of protocol identifier is given by the string: "`ip.src`".

Subfield Identifiers

A subfield identifier handles the case of nested fields, where different portions of the same field can have their own name: such nesting can have arbitrary depth[1]. Subfields are represented as a recursive sequence of field identifiers separated by a "`.`" (dot):

proto-name . *field-name* . *sub-field-name* . *etc.*

where *sub-field-name* is the name of a portion of the field identified by *field-name*.

It should be noted that even if a field is actually a portion of another field, in most cases there is no need to specify the entire field hierarchy that leads to it. In other words, *sub-field-name* in the previous example can also be referred with the more compact representation:

proto-name . *sub-field-name*

The field hierarchy must be explicitly specified only in those cases where two or more fields with different names share a nested field with the same name.

An example of protocol with subfield identifiers is TCP, which contains the `flags` field, which in turn contains the subfields named `syn`, `ack` and so on. In that case the `syn` field can be identified by the string "`tcp.flags.syn`"; or, alternatively, by "`tcp.syn`", as in TCP there is no other field that contains a subfield named "`syn`".

Virtual Fields

A *virtual field* is an entity that does not refer to an actual field specified in a protocol description, but to a given portion of the packet, and can be accessed in the same way as normal protocol fields by using the notation:

proto-name . *virtual-field*

Currently, NetPFL defines three virtual fields:

1. `header`: the portion of the packet containing the header of the specified protocol. For example: `tcp.header`.

2. `payload`: the set of data following the specified protocol header. This may include also the header of subsequent protocols. For example, in the case of a TCP/IP

[1] However, having more than two levels of nested field names is not common in NetPDL

packet, `ip.payload` includes all the bytes that are present *after* the IP header until the end of the packet buffer[2].

3. `data:` all the bytes from the beginning of the given protocol to the end of the packet buffer. In general, this is equivalent to have the protocol header plus the payload, but it is implementation-dependent (see note above). For example, `tcp.data` includes the layer-IV TCP header and its payload.

The `frame` placeholder

The `frame` keyword represents a reference to the whole packet buffer. It is a shortcut for

current_link_layer_protocol`.data`

where `current_link_layer_protocol` is the layer-II protocol used on the link on which the filter is running, and `data` is the virtual field described above. This keyword enables to refer to the packet as a whole, without referring to any specific link-layer header.

The `packet` placeholder

The `packet` keyword represents a reference to the layer-III header and payload possibly present in a network frame. It is a shortcut for

current_layer3_protocol`.data`

where `current_link_layer3_protocol` is the layer-III protocol found in the current frame, and `data` is its virtual field. The protocols actually supported for the `packet` placeholder are dependent on the specific implementation of NetPFL.

1.5.2 Byte Ranges

Byte ranges can be used to operate on a portion of the packet, identified by a sequence of consecutive bytes, and are specified using the notation:

identifier`[`*offset*`:`*length*`]`

Where:

- *identifier* can be a protocol identifier, a field identifier, a virtual field, or either the `frame` or `packet` placeholders;

[2]Please note that this definition is somewhat implementation-dependent. For instance, if the packet buffer contains the Ethernet frame, the IP packet, and the Ethernet CRC trailer, `ip.payload` will contain not only the bytes until the end of the IP packet, but also the Ethernet CRC bytes, i.e. all the bytes till the end of the packet buffer.

- *offset* is the displacement of the first byte of the sequence from the entity we are referring to (i.e., the one specified by *identifier*);
- *length* is the length in bytes of the sequence

If *length* is omitted, its default value is 1.

Examples:

- `frame[12:2]` identifies a sequence of two bytes starting at offset 12 of the packet buffer
- `ip[16:4]` identifies a sequence of 4 bytes starting at offset 16 from the beginning of the `ip` header (i.e., the ip destination address)
- `ip.src[0:2]` identifies a sequence of two bytes starting at offset 0 from the beginning of the `ip.src` field
- `tcp.header[2:2]` identifies a sequence of two bytes starting at offset 2 from the beginning of the `tcp` header, (i.e., the tcp destination port). Note: it is equivalent to `tcp[2:2]`
- `tcp.payload[0:3]` identifies a sequence of three bytes starting at offset 0 after the end of the `tcp` header

1.5.3 Data References

NetPFL provides several methods for identifying a portion of network data, such as using field identifiers, virtual fields, byte-ranges, or the `frame` and `packet` placeholders. These elements are called *data references*, as they allow to *refer* to a portion of the packet buffer.

Among *data references*, we distinguish between:

- *simple data reference*: a data reference created through a byte-aligned field identifier[3], virtual field, byte range or the `frame` and `packet` placeholders.
- *bitfield data reference*: a data reference created through a bitfield, i.e. a field whose starting and/or ending offset are not aligned to an integer number of bytes.

Size of data references

Simple data references are treated by NetPFL as sequences of bytes. This requires any `rvalue` present in an expression (Section 1.6.2) to be a byte sequence.

However, in some cases a simple data reference can be treated as an integer value, in particular in the following cases:

[3]When the NetPDL language is used to define protocols and fields, a *simple data reference* is obtained when referring to any type of field but the `bit` field.

1. Fields of fixed size (i.e., whose size is known at compile-time), whose length is less or equal than 4
2. Byte-ranges whose length is less or equal than 4

In any other case, i.e. when the length of a reference is greater than 4 bytes, or its length is known only at run-time, the packet reference is **always** evaluated as a byte array. It should be noted that when treating a reference as a 32 bit number, an implicit byte order conversion from network byte order (i.e. big-endian, the conventional byte order used in network packets) to host byte order is applied.

1.5.4 Constant Values

Due to its network-oriented nature, NetPFL handles primarily two data types: *unsigned 32-bits integers* and *byte sequences* (or byte arrays). Additional types are *regular expressions* and *field-dependent values*.

Unsigned 32-bits integers

The possible ways to express constant numeric values are the following:

1. Unsigned decimal values on 32 bits, e.g. `1258`
2. Hexadecimal values on 32 bits, using case-insensitive C-like notation, e.g. `0x1096ccFF`
3. IP addresses, using the dotted notation, e.g. `192.168.0.1`

Byte sequences

The possible ways to express constant byte sequences are the following:

1. String literals enclosed in double quotation marks, e.g. "`HTTP`"
2. Sequences of hexadecimal bytes, in which consecutive bytes are separated by a "`:`" (colon), e.g. `AA:bb:CC:dd`. Please note that Ethernet MAC addresses are a special case of six bytes long hexadecimal byte sequences.
3. IPv6 addresses, e.g. `2001:0db8:0020:ABCD:efff:0000:1428:57ab`

Regular expressions

A regular expression is a string enclosed between double quotation marks (e.g. ".*HTTP GET.*"). From a syntactical point of view, a regular expression is a string that includes one or more metacharacters, e.g. ".", "*" etc. However, as will be detailed in Section 1.6.2, strings and regular expression are used with different kinds of operators, thus avoiding any ambiguity. Regular expressions allowed in NetPFL follow the PERL dialect.[4]

[4]The PERL syntax is the one implemented in the well-known `libpcre` library, and it is more powerful albeit more complex.

Field-dependent values

A field-dependent value is a string, enclosed by single quotation marks, which is mapped to an integer or a byte array. This data type is used as operand in comparison expressions. The mapping between the string and the real value depends on the protocol field to which the field-specific value is compared. For example, the predicate

```
ip.src == 'www.polito.it'
```

compares the `src` field of an IP header with the field-specific value *'www.polito.it'* (the syntax of comparison expressions will be explained in details in Section 1.6.2). In this case, field-specific constant is translated into an IPv4 address through a DNS lookup. In this other example:

```
tcp.sport == 'http'
```

the predicate compares the `sport` field in the TCP header with a field-specific value. In this case, *'http'* is translated into a TCP port value through a table that associates well-known services with their default ports.

The methods used for mapping keywords to actual values are not specified and are left to the specific implementation.

1.6 Filtering Expressions

A filtering expression is a Boolean function representing a (possibly complex) condition on network data that must be met for the action of a NetPFL rule to be triggered. This section introduces the basic syntax and semantic for filtering expressions, without taking into account problems related to complex encapsulation cases which could be present in packets (such as those involving tunneling). Section 1.9 describes a set of additional primitives of the NetPFL language that will allow a correct handling of packets containing tunneled encapsulations.

1.6.1 Basic Syntax

The elementary blocks of NetPFL filtering expressions are represented by *predicates*, which are Boolean functions representing simple conditions on the content of a packet. Complex filtering expressions can be expressed by joining together several basic predicates, as will be outlined in Section 1.6.3.

Simple predicates are expressions that check the presence of a protocol or a field within the packet.

Predicates on Protocols

The simplest kind of filtering expression is the one specifying only a protocol identifier, with the following meaning:

if at least an instance of `protocol` *is present in the packet, then the predicate is true*

An example is "`ip`". Please note that this predicate is satisfied when an `ip` header is found in any position of the packet, e.g. also in case of an `IPv6` packet transporting an `IPv4` packet inside. More details will be shown in Section 1.6.4.

Predicates on Header Fields

Predicates on Header Fields are similar to the previous case. For instance, a condition consisting of a field identifier, such as "`ip.timestampoption`" will define a filter that selects all the packets in which this field is present.

This notation might appear useless when it refers to a mandatory field of a protocol: for example, the predicate "`ip.src`" always matches all IP packets, since the `src` field is always present in the IP header. However, this may be useful to select packets in which a given field may appear (optionally) within a protocol header only under some circumstances.

Section 1.6.4 will present the case in which a field is repeated multiple times within the packet.

1.6.2 Conditional Predicates

As seen in Section 1.5.3, the content of a network frame is accessible through data references, i.e. fields, virtual fields, byte-ranges and the `frame` and `packet` placeholders. We can distinguish two types of conditions: simple conditions through standard comparison operators and content search operators.

Standard comparison operators

A predicate can represent the evaluation of a condition on a data reference, through comparison operators, with the following syntax:

reference `CompOperator` *constant*

or

reference1 `CompOperator` *reference2*

`CompOperator` is an operator from Table 1.1.

In the case of a comparison with a constant, this can be specified using one of the syntactical constructs defined in Section 1.5.4. In case a numeric constant is being used, please refer to the rules listed in Section 1.5.3. Briefly:

1. The constant value **can be** of integer type only if the reference can be evaluated as an integer (refer to 1.5.3 for more details)

Operator	Meaning
==	equal
!=	not equal
>	greater than
<	less than
>=	greater or equal
<=	less or equal

Table 1.1. Comparison operators

2. The constant value **must be** of integer type if the reference is a bitfield

3. The constant value **must be** a byte-sequence if the reference cannot be evaluated as an integer, i.e. its length is greater than 4 bytes or is unknown

4. The constant value **can be** a byte-sequence even if the reference can be evaluated as an integer

When comparing against an integer constant, a comparison between unsigned integers is applied; on the other hand, when comparing against a byte-array constant, a lexicographical comparison is applied. When comparing two packet references, an integer comparison is applied whenever possible (i.e. when both references are known to be smaller or equal to a 32 bit integer), while a lexicographical comparison is used in any other case.

Content search operators

Besides comparisons, predicates may involve content search operations (listed in Table 1.2). During a content search an operand is examined to determine if it contains a specific sequence of bytes. The following syntax is used:

reference `(matches | !matches)` *string-const*
or
reference `(contains | !contains)` *string-const*
or
reference1 `(contains | !contains)` *reference2*

The following rules apply:

1. Only *simple data references* are supported (hence, *bitfield data references* cannot appear in a content-search string)

2. When using `contains` and `!contains` operators the constant string (*string-const*) is interpreted as a simple sequence of characters and exact pattern matching is used, while when using `matches` and `!matches` operators the (*string-const*) is interpreted as a PCRE-compatible regular expression.

Operator	Meaning
`matches`	The predicate is true if the `lvalue` matches the `rvalue` expression.
!matches	The predicate is true if `lvalue` does *not* match the `rvalue` expression.
`contains`	The predicate is true if `lvalue` contains the `rvalue` expression.
!contains	The predicate is true if `lvalue` does *not* contain the `rvalue` expression.

Table 1.2. Content-search operators

Operations before comparison

When a packet reference can be treated as an integer, it can be the operand of an arithmetic or bitwise logic expression. NetPFL allows a wide set of operations, involving both references and integer constants. Available operations follow the syntax and the grammar of the C-language, and are listed in Table 1.3.

All the operators but "∼" are binary operators. Vice versa, "∼" is a unary operator.

Examples of statements in which a predicate consists of a comparison between a header field and a constant value can be found in table 1.5. In the last three examples, various operations are performed on the first operand.

Arithmetic and logic operators listed in Table 1.3 have higher precedence over other operators (e.g. Table 1.1 and Table 1.2).

The precedence rules for those operators are defined (in descending order of precedence) in Table 1.4. In case several arithmetic and logic operators are present in the same expression, precedence is given according to these rules: *(i)* operators higher in Table 1.4 have a higher precedence, while *(ii)* operators on the same line in the list have the same precedence. In the latter case, if several operators are present in the same expression, these are evaluated left-to-right. Parentheses can be used to change the precedence within a single expression.

Operator	Meaning	Type
+	Add	Arithmetic operator
-	Sub	Arithmetic operator
*	Multiply	Arithmetic operator
&	Bitwise AND	Bitwise operator
|	Bitwise OR	Bitwise operator
~	Bitwise NOT	Bitwise operator
<<	Left shift	Shift operator
>>	Right shift	Shift operator

Table 1.3. Arithmetic operators

~
*
+ -
<< >>
&
|

Table 1.4. Precedence rules for arithmetic and logic operators

1.6.3 Predicate Composition

In order to express complex conditions, basic predicates can be joined together as follows:

predicate1 `BoolOp` *predicate1*
or
`not` *predicate*

`BoolOp` can be either `and` or `or`. As expected, the compound predicate resulting from an `and` operation is true only if both predicates are true, and is false in any other case; vice-versa, the result of an `or` operation between two predicates is false only when both the sub-conditions are false, and true in any other case. A predicate can also be negated

Expression	Meaning
`ethernet.dst == aa:bb:cc:dd:ee:ff`	**true** if ethernet is present and the destination MAC address is equal to aa:bb:cc:dd:ee:ff
`udp.dPort == 53`	**true** if UDP is present and the value of the UDP header field *dPort* is equal to 53
`ip.dst != 10.0.0.1`	**true** if IPv4 is present and the value of the IP header field *dst* is not 10.0.0.1
`ip.src & 255.255.255.0 == 10.0.0.0`	**true** if the ip source address masked with 255.255.255.0 is equal to 10.0.0.0
`ip.src & 0xFFFFFF00 == 10.0.0.0`	**true** if the ip source address masked with 0xFFFFFF00 is equal to 10.0.0.0.
`tcp.payload matches "GET.*HTTP/1.(0|1)"`	**true** if the regular expression is found in the specified virtual field.
`ip.src & 255.255.255.0 == ip.dst & 255.255.255.0`	**true** if the first 24 bits of the IP source address are equal to the first 24 bits of the IP destination address.
`ip.hlen * 5 == 20`	**true** if the IPv4 header length is 20 bytes.

Table 1.5. Examples of expressions

through the `not` operator.

NetPFL allows the use of C Language's boolean operators in place of their literal counterparts, i.e. `&&` instead of `and`, || instead of `or` and `!` instead of `not`.

Precedence and associativity

Operators used for predicate compositions have always a lower precedence than arithmetic and logic operators (Section 1.6.2).

In case several conditions are present within the same predicate, precedence is given as shown in the list below (operators higher in the list have a higher precedence):

```
not
and
or
```

When multiple boolean operators are used within the same filter, parentheses can be used to make the precedence between operators explicit. Table 1.6 shows some examples of expressions in which boolean operators are used.

Expression
`(ip and tcp.dPort >= 1024) or (ip.src == 10.0.0.7 and udp)`
`(ip or ipv6) and tcp`
`not ip.src == 10.0.0.7`

Table 1.6. Examples of composed boolean expressions

A note about the operators `not` and `!=`

It is worth noting that the operators `not` and `!=` have a slightly different meaning, and, if used interchangeably, lead to predicates with different meanings.

For instance, the filter `not ip.src == 10.0.0.7` is *not* equivalent to `ip.src != 10.0.0.7`. In particular,

`not ip.src == 10.0.0.7`

should be interpreted as:

`not(`*the packet contains* `ip` *and the field* `ip.src` *is equal to* `10.0.0.7)`

Which means:

either the packet does not contain `ip` *, or (the packet contains* `ip` and the field `ip.src` *is not equal to* `10.0.0.7)`

On the other hand,

`ip.src != 10.0.0.7`

should be interpreted as:

the packet contains `ip and` *the field* `ip.src` *is* `not` *equal to* `10.0.0.7`

In other words, the filter `not ip.src == 10.0.0.7` accepts all the packets containing `ip` with `ip.src != 10.0.0.7`, as well as all the packets not containing `ip`. On the other hand, the filter `ip.src != 10.0.0.7` accepts only the packets containing `ip` with `src != 10.0.0.7`.

The same statement applies to operators `!matches` and `!contains`.

Figure 1.1. Packet containing an ip-gre-ip tunnel

1.6.4 Multiple instances of the same protocol/field

In case multiple instances of the same protocol/field are present within the packet, the filtering process will check the filter against all instances until a match is found; this represents the default behavior of the language. If we want instead to specify more precisely which instance we want to consider, new mechanisms have to be introduced. Particularly, we can distinguish two cases: the one in which the filtering predicate is a protocol and the one in which the filtering predicate involves protocol fields.

The filter predicate is a protocol

In this case, the filter returns `true` if the protocol is found anywhere within the packet. In order to enable the selection of the instance of the protocol we are referring to, a new mechanism has been introduced: *header indexing* allows to specify exactly which protocol header instance we are interested in. The syntax is as follows:

```
proto_id%n
```

where `n` is an integer number indicating the ordinal number of the occurrence of the `proto_id` protocol to be taken into account. Different instances of the same protocol header are numbered with increasing values of `n`, where `proto_id%1` represents the first occurrence of the `proto_id` protocol header.

For example, considering the packet depicted in Figure 1.1, the expression

```
ip%1
```

refers to the first instance of the `ip` protocol (i.e. the one between `ethernet` and `gre`), while the expression

```
ip%2
```

refers to the second instance of the `ip` protocol (i.e. the one between `gre` and `tcp`).

The parameter n can also be specified by using the `inner` keyword, which allows selecting the innermost (i.e. the last) instance of the `proto_id` protocol. For example, always taking into account the packet exemplified in Figure 1.1, the expression

```
ip%inner
```

selects the second instance of the `ip` protocol.

Header indexing can be used for specifying more selective filtering conditions involving protocols and fields. In particular, since the filtering condition

```
ip%2
```

matches the second instance of the `ip` protocol, it can be used to match all packets containing **at least two** instances of the `ip` protocol header. Furthermore, the filtering condition

```
ip%1.src == 10.0.0.1
```

matches only the packets where the `src` field of the **first** instance of the `ip` protocol equals the value `10.0.0.1`.

1.7 Actions

The *action* defines what to do when a packet matches the filter.

NetPFL defines the following actions:

- `returnpacket`
- `extractfields`

The former can be used to accept a packet, while the latter allows to extract and return to the user a list of fields (or, in the general case, a list of packet references, as well as additional information about the content of a packet). The result of an action, i.e. the data returned to the user, is action-dependent.

1.7.1 Accepting Packets

The `returnpacket` action is self-explicative, and does not need any additional considerations. If the input packet satisfies the filter, the packet itself is returned to the user. The return format of the packet is application-dependent and it is not specified by the NetPFL.

If no other action is specified, `returnpacket` is the default one.

1.7.2 Extracting network data

The `extractfields` action allows the user to specify a list of data references (Section 1.5.3) and/or some additional information to be extracted and returned to the user, as metadata associated to the packet, if the input packet satisfies the filter. The format of the returned data is implementation-dependent.

The basic syntax for the `extractfields` action is the following:

`extractfields(` *list-item-1* `,` *list-item-2* `,` *...* `,` *list-item-n* `)`

Where *list-item-1* `...` *list-item-n* can be one of the following:

1. a *data reference*: returns the content of a field, a virtual field, a byte-range. Please refer to Section 1.5 for a definition of packet reference.
2. `protolist`: returns the *protocol IDs* of the protocols encountered during packet demultiplexing (i.e., a packet containing *ethernet-IP-TCP-HTTP* protocols will return the list of protocol IDs *ethernet, IP, TCP, HTTP*).
3. `innerproto`: returns the *protocol ID* of the innermost protocol (i.e., a packet containing *ethernet-IP-TCP-HTTP* will return *HTTP*).
4. *proto*`.allfields`: if *proto* is present, returns the content of each field contained in the specified protocol header. Since the fields actually present in the header may be known only at runtime, the *field ID* of each field must be returned as well.

Protocol IDs and *Field IDs* are application-dependent. The NetPFL implementation must provide the appropriate methods to interpret the returned IDs.

The `extractfields` action leads to some corner-case behaviors that are listed below.

Multiple instances of the same field

Filtering and *action* are two consecutive phases in the processing path. The filtering phase returns a set of packets, which are then further processed in order to extract the desired data. Section 1.6.4 analyzes the case of multiple instances of protocols/fields within the same packet with respect to packet filtering. Here we present the extraction rules when multiple instances of the same protocol/field are present within a packet.

In case multiple instances of the same protocol/field are present within the same packet, by default the first instance that is found is returned. However, in order to select more precisely which instances have to be returned, the same indexing techniques that have been introduced for the filtering process can be used in the extraction process. A more complex list of examples can be found in Section 1.9.

Absence of a field within the current packet

If may happen that a field contained in the field list is not present in the packet. This may be the case of an optional field, or the case of a filter that extracts fields that are not present in all matching packets (e.g. all packets are matched, but only `ip.src` is extracted: not all matching packets might include an instance of IP). In this case, NetPFL implementations can either return a *null* reference, or a field with zero-length, or skip this field from the returned metadata.

Format and order of the returned metadata

The format and the order of the returned metadata is implementation-dependent. Therefore, the user should make no assumptions.

1.8 Streams

A `stream` can be viewed as a sequence of processed packets with the corresponding metadata. With the definition of the stream, the packet processing engine can handle several processing programs at the same time, with possible optimizations between the different streams (e.g., a test required by all active streams may be performed only once).

From the end-user point of view, a stream may be useful at least in these conditions:

1. **multiple matching**: in case of a packet matching multiple streams, we can immediately know which ones returned `true`.
2. **editing**: we can easily add a new stream, delete an existing stream, or replace an existing one without any modification to the rest of the active streams.

The NetPFL implementation has to provide the equivalent of the `setstream()` and `deletestream()` functions. If a new stream is added through the `setstream()` function, the NetPFL engine will check if the new stream has to replace an existing one or has to be added to the existing stream set. This control is based on the *stream ID*: if this value matches an existing stream, the old stream has to be replaced, otherwise the new stream has to be added.

In case no streams are specified, the NetPFL engine will assume the stream *0* as default.

The set of functions required to handle streams (and to get results from multiple streams at once) are implementation-dependent.

1.9 Handling tunneling in the encapsulation graph

Consider a set of protocols; the encapsulation relationships that exist between them can be used to identify a directed graph $G(V, A)$, where each node v represents a protocol in the database, and an edge $e(x, y)$ is directed from the node x to the node y if the protocol y can be encapsulated into the protocol x. We call such a graph a *Protocol Encapsulation Graph*, or *Encapsulation Graph*.

While most of the protocols are characterized by a small set of simple encapsulation rules, the complete set of possible encapsulation rules is far larger. Figure 1.2 shows an indicative example.

With the NetPFL syntax described so far, a filter involving a protocol will cause the NetPFL engine to control all the possible encapsulation paths that lead to that protocol. For instance, in case the `ip` protocol is considered, NetPFL will evaluate *ethernet-ip*, *ethernet-vlan-ip*, *ethernet-ipv6-ip*, and even *ethernet-ipv6-ipv6-ip*, and more, until (possibly) an instance of `ip` is found.

This section will present a set of additional primitives for the NetPFL language that allow selecting the encapsulation paths that have to be considered when checking for a protocol. These new keywords operate by modifying the encapsulation graph, enabling only a subset of the paths that terminate on the desired protocol. This leads to the creation of simpler and more compact filtering programs.

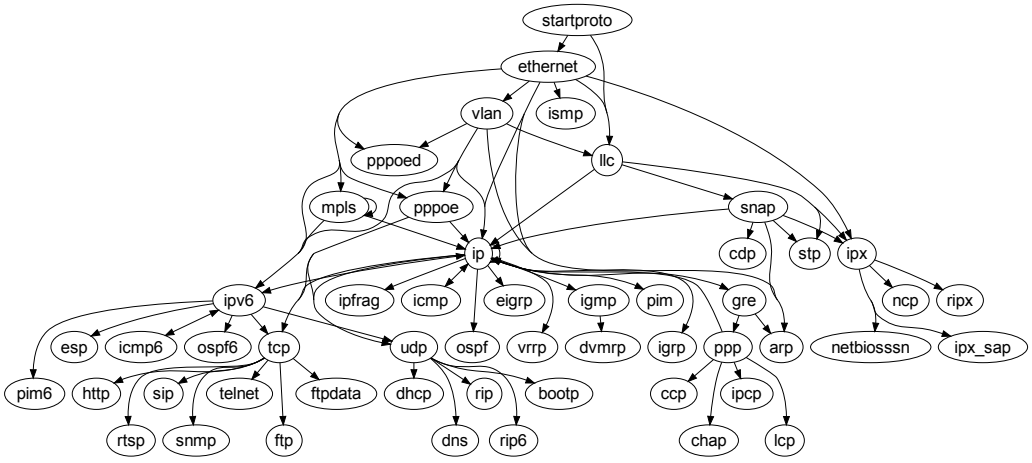


Figure 1.2. Encapsulation Graph

In addition, the new keyword `tunnel` is introduced, which enables to capture only the traffic in which the selected protocol is tunneled. For instance, if we consider two packets, respectively carrying `ethernet-ipv4-ipv6-tcp` and `ethernet-ipv6-ipv4-tcp`, we can decide to capture either:

- the packet in which the IPv6 header is not tunneled (`ipv6 notunnel`), which would select the second packet;
- the packet in which the IPv6 header is tunneled (`ipv6 tunnel`), which would select the first packet;
- all the packets that have an IPv6 header (`ipv6`), which would select both packets.

Please consider that protocols `ethernet` and `tcp` in those packets are **not** tunneled. This example emphasizes the fact that tunneling can be identified only if each protocol is associated with the right layer of the network stack. Hence, we should envision an algorithm that assigns the proper layer to all the protocols in the encapsulation graph.

1.10 Identifying Protocol Header Instances

In order to correctly handle packets containing tunneled encapsulations, the user must be given the possibility to specify more selective filtering conditions, with respect to those that can be defined with the primitives presented in previous sections. In particular, if tunneling is used, a packet might contain multiple headers belonging to the same protocol. For example, a MPLS label may encapsulate another MPLS label which in turn can encapsulate another one and so on.

In this paragraph, we introduce the concept of *header instances*, which are the basic building blocks over which we will build the more complex constructs described in the following sections.

1.10.1 Definition of header instances

We give here a more formal definition of what header instances are.

- The term *Header Instance* is defined by either:
  - a condition selecting *Any Instance* of a specific protocol
  - a condition selecting an *Indexed Instance* of a specific protocol
- The term *Any Instance* is defined by either:
  - a protocol identifier (i.e. any header of the specified protocol)
  - a condition involving fields of a **single** protocol (i.e. any header of the specified protocol, where the condition on its fields is verified)
- The term *Indexed Instance* is defined by either:
  - an *Indexed Protocol* (i.e. the n-th header of the specified protocol)
  - a condition involving fields of a **single** *Indexed Protocol* (i.e. the n-th header of the specified protocol, if the condition on its fields is verified)
- The term *Indexed Protocol* is defined either by:
  - a protocol identifier followed by the '%' sign, followed by an integer number: `proto_id%n`
  - a protocol identifier followed by the '%' sign, followed by the `inner` keyword `proto_id%inner`

In order to avoid ambiguities and undefined behaviors, NetPFL does not allow basic condition predicates to be based on fields of different protocols. This implies that, when a condition involving protocol fields is specified through the relational operators listed in Table 1.1, every referred field must belong to the same protocol.

For example, the condition:

```
ip.src & 255.255.255.0 == ip.dst & 255.255.255.0
```

is perfectly legal, while a condition like:

```
udp.dport == tcp.dport
```

is not allowed.

1.11 Header Sequences

In this section we introduce constructs and building blocks based on the presence of particular sequences of protocol headers.

1.11.1 The `any` keyword and other *protocol placeholders*

`any`

The `any` keyword represents a wildcard matching any protocol defined in the database in use. Its purpose will be clarified in the following, however it can be thought of as a "*don't care*" placeholder.

1.11.2 Repeat operators

Repeat operators are used to describe situations in which a particular header may occur a variable number of times in packets. The paradigm and the syntax are borrowed from the quantifiers used in the regular expressions. In particular, *repeat operators* allow specifying a *header sequence* as:

`proto_id<RepOp>`

Where `proto_id` is a protocol identifier, or the `any` keyword, and `RepOp` is one of the symbols from Table 1.7.

Operator	Meaning
?	The question mark indicates zero or one consecutive occurrences of a protocol header
*	The asterisk indicates zero or more consecutive occurrences of a protocol header
+	The plus sign indicates one or more consecutive occurrences of a protocol header

Table 1.7. Repeat operators

For Example:

`ip+`

allows matching all the packets that contain one or more consecutive instances of the `ip` protocol, while:

`mpls*`

matches all the packets that contain zero or more consecutive instances of the `mpls` protocol.
A special case is represented by:

`any?`

```
any*
any+
```

that always match all packets (which are supposed to contain at least one header).

As we can see, the `any` keyword used with repeat operators is not meaningful if taken alone, however it plays an important role for specifying *header chains*, which are described in the following section.

1.11.3 Chaining headers through the "in" and "notin" operators

The **in** and **notin** keywords allow the user to specify an exact sequence of heterogeneous headers that must be found in a packet in order to have a match. In the following we refer to these two keywords as *Chaining Operators* (`ChainOp`s).

We recursively define a *Header Chain* as:

- a single *Chain Element* (described later)
- a sequence *Header Instance* `<ChainOp>` *Header Chain*
- a sequence *Header Chain* `<ChainOp> tunnel`
- a sequence *Header Chain* `<ChainOp> defaultencap`

where a *Chain Element* can be either:

- a *Header Instance* as described in Section 1.10
- a *Protocol Placeholder* (i.e. the `any` or `layer <n>` keywords)
- a protocol identifier or the `any` keyword followed by a repeat operator (`proto_id<RepOp>`, or `any<RepOp>`)
- a *Header Set* (described in Section 1.11.3)
- a *Header Set* followed by a repeat operator

The `in` operator allows defining a chain where the left-hand element is `directly` encapsulated in the right-hand element. The `notin` operator is the dual of `in` and allows specifying a chain where the left-hand element is `directly` encapsulated in any header other than the one defined by the right-hand element. For instance, by specifying

```
ip in vlan
```

we want to match all the packets containing an `ip` header encapsulated in a `vlan` one, while specifying

```
tcp notin ip
```

we want to match all the packets containing a `tcp` header encapsulated in any header other than `ip` (e.g. `ipv6`).

In order to clarify the concept of *header chains* we present here some basic examples:

`ipv6 in ip`
matches all the packets with an `ipv6 in ip` tunnel

`tcp in any in ip`
matches all the packets where `tcp` is encapsulated in any protocol, again encapsulated in `ip`

`udp in ip+`
matches all `udp` packets encapsulated in a sequence of at least one `ip` header

`ip.src == 10.0.0.1 in vlan+ in ethernet`
matches all `ip` packets where `src == 10.0.0.1`, encapsulated in a sequence of at least one `vlan` header, again encapsulated in `ethernet`

`ip in vlan* in ethernet`
matches all `ip` packets, encapsulated in a sequence of zero or more `vlan` headers, again encapsulated in `ethernet`

Header Sets

A header set is represented by a group of one or more *header instances* and it enables additional flexibility in the definition of *header chains*, by allowing to specify more precisely the demultiplexing paths to be taken into account. In particular a header set is simply a comma-separated list of *header instance* definitions, enclosed in curly braces:
{*HeaderInst-1*, *HeaderInst-2*, ..., *HeaderInst-n*}

Some examples of header sets are:

- {`ip, ipv6, arp`} i.e. select any ip header, or any ipv6 header, or any arp header in the packet
- {`tcp.dport == 80, tcp.sport == 80`} i.e. select any `tcp` header where `dport == 80`, or any `tcp` header where `sport == 80`
- {`ip%1, ipv6%1`} i.e. select the first instance of the `ip` and `ipv6` protocols

As previously mentioned, *header sets* can be used as elements of a *header chain*, allowing to specify very complex matching conditions, as the following one:

`skype in {tcp, udp} in any+ notin vlan*`

that matches all the packets belonging to a `skype` session, either using `tcp` or `udp` transport, encapsulated in any sequence of headers (tunneled or not), except if encapsulated in any sequence of `vlan` headers.

1.11.4 Contexts

The paradigm of *tunneling contexts* allows to deal with tunneling without explicitly specifying a protocol chain. Contexts are less flexible than protocol chains, but are more powerful when writing some types of filters.

The concept of "context" is fairly simple:

- The sequence of headers in a packet is divided in one or more consecutive contexts.
- The first context starts at the beginning of the packet. A new context starts whenever a tunnel is detected, i.e. each time the layer of a protocol is less or equal than the layer of the protocol that encapsulates it.

Figure 1.3 depicts how a packet is subdivided in contexts. The Ethernet header and the first IP header belong to context #1. The presence of a second IP header after the first one causes another context to start.

The key concept behind contexts is that, in many cases, a set of conditions is "interesting" only if they are all verified within the same context. For example, if a network administrator wants to intercept all the http traffic coming from a given server, it is not important if such traffic is tunneled or not. NetPFL allows to manage such situations by specifying that all the conditions belonging to a set must be verified within the same context.

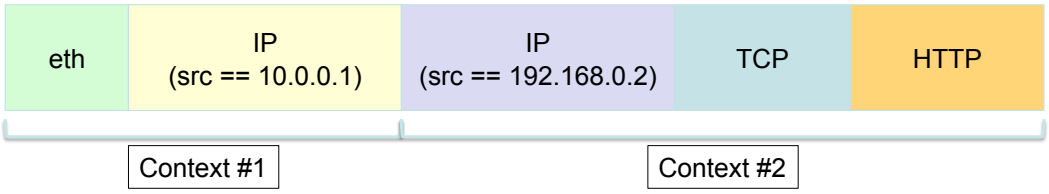


Figure 1.3. Contexts example

All conditions that must be verified within the same context must be enclosed between "<" and ">":

```
<set_of_conditions>
```

For example, the following filter matches all packets carrying `http` traffic coming from 192.168.1.3, either if it is encapsulated or not:

```
<ip.src == 192.168.1.3 and http>
```

Note that neither protocol chains nor protocol sets are allowed between the context operators.

1.12 Conclusions

We have presented NetPFL, a new packet filtering language, which can naturally support complex situations of tunneled and stacked encapsulations. This feature is becoming more and more important because the number of possible protocol encapsulations in network traffic is growing day after day, so we need to inspect also tunneled traffic and/or to control precisely which encapsulations we are referring to.

Additionally, the *filter-action-stream* model allows to configure thoroughly the behavior of the filter, giving the user a more precise control over the dynamics of a filtering expression and extending the operations done (efficiently) in the packet filter without the necessity to deliver all the packets to the application. For example, this model supports several independent filters to be deployed on the same datastream, thus allowing multiple matching of different conditions; the user can also configure which action has to be taken whenever a filtering condition is satisfied.

Furthermore, another structural choice that enhanced the flexibility of the architecture is the separation between the filtering component and the protocol description one: in fact, while in our implementation NetPFL has been used in conjunction with NetPDL, any kind of protocol description language can be adapted to be deployed, since NetPFL does not contain in itself any knowledge regarding protocols, headers and fields.

While NetPFL covers the issues w.r.t. filtering expressivity, there are still open questions about the efficient implementation of a packet filter. The answers to those questions will be addressed in the next chapters.

Part II

About efficient packet filters

Chapter 2

pFSA: a new model for packet filters

2.1 Introduction

In the recent years, we witnessed many changes in the computing world. The network interaction among devices has evolved significantly, so we had to re-engineer our computer networks in order to accommodate many new use cases. A large portion of these requirements introduced new features in the network protocols stack, whose complexity increased as a consequence. For instance, low-level network protocols are growing in number: new solutions, arising in particular for the purpose of network virtualization (e.g., 802QinQ, VXLAN), are rapidly transforming our Ethernet frames [22]. The middle layers of the protocol stack are facing a similar metamorphosis: examples include the widespread adoption of Virtual Private Networks with their bizarre tunneling mechanisms, the necessity to transport IPv6 traffic over IPv4 networks (with different encapsulation methods, such as pure IPv6 encapsulation in IPv4, or through GRE or even UDP, and more) and WAN traffic transports.

Packet filtering represents a niche that may be dramatically affected by those changes. Packet filters are the basic building block of many applications, such as firewalls, network monitors and more, and the capability to capture at high speed the traffic we want, independently from the lower level encapsulations, is becoming more critical day after day.

This chapter presents **pFSA**, a new model for packet filtering that ensures the optimal number of checks on the packet in order to take the matching/not-matching decision. This result is obtained by transforming packet filtering rules into Finite State Automata (FSA), which guarantee optimal results even in case of multiple filters combined together. Vice versa, the *ad hoc* optimization techniques used by most of the previous approaches are based on heuristics that cannot provide such guarantees, which are needed to ensure the best performance when operating in the conditions mentioned before (multiple filters or unconventional encapsulations). Furthermore, our model is generic enough so that it does not require *a priori* protocol definitions: in our prototype the protocol database is provided

at run-time and it can be easily extended or modified in order to recognize any protocol or encapsulation the user is interested in. This means that we can create the best filtering automaton whatever encapsulation we may have, including unusual protocol patterns such as tunneling (and self-tunneling) of any kind; the generated filter is able to locate the desired pattern in the network traffic, independently from the actual protocol stack of the given packet. Finally, we present also a prototype implementation that translates the pFSA model into running code, although this step cannot formally maintain the optimality properties guaranteed by the model.

Section 2.2 presents the state of the art; Section 2.3 introduces the proposed model; Section 2.4 presents the application of that model to the packet filtering domain, while Section 2.5 is dedicated to the problem of optimization on protocol fields. Finally, Section 2.6 presents an overview of our implementation, leaving the experimental evaluation to Section 2.7 and conclusions to Section 2.8.

2.2 Related Work

The **CMU/Stanford Packet Filter** [23] (CSPF) represents the ancestor of any modern packet filter. It introduced the concept of a kernel-level virtual machine that executes an application-provided program (i.e., the packet filter), which can be defined at run-time. However, its optimizations capabilities were limited.

The **Berkeley Packet Filter** [8] (BPF) is also based on a virtual machine and brings some notable improvements, such as the adoption of the *Control Flow Graph* model, which enables the deployment of compiler techniques to remove redundant checks from the generated code. The BPF model was later improved by **BPF+** [11], which uses even more aggressive optimizations derived from software compilation techniques and adds a Just-In-Time (JIT) compiler.

PathFinder [9] adds the possibility to compact the Control Flow Graphs of different filters. Each expression in a filter is exploded into a list of *cells*, each one describing a step in the construction of the final check; equivalent cells coming from multiple filters *may* be merged together. However, filters are optimized only if they share a common prefix; for instance, `tcp.sport` is always checked twice in the expression `(tcp.sport == X and tcp.dport == W) or (ip.src == K and tcp.sport == Y)`.

The **Dynamic Packet Filter** [10] (DPF) extends the previous approach by introducing the capability to generate native code instead of running the filter into an interpreter. Furthermore, field coalescing is introduced, allowing fields at contiguous offsets to be checked together; for example a single 32-bits check against the word `0x00800090` is performed for the filter `tcp.sport == 0x80 and tcp.dport == 0x90`.

The recently proposed **Stateless FSA-based Packet Filter** (SPAF) [13] exploits Finite State Automata for packet filter generation and guarantees, by construction, code optimality and safety. Each protocol is modeled through a byte-consuming automaton, which reads the bytes that are part of the protocol and follows the encapsulation rules (e.g., the starting state of the `IP` protocol is linked to the exit state of the `Ethernet` protocol when the bytes associated with the `EtherType` field have the proper value);

different automata are then joined together using the algorithms known from the literature. However, SPAF is extremely slow in the automata generation phase, because the protocol field abstraction is lost very early in the computation, hence the amount of generated states tends to be rather high. This has a huge impact on FSA construction, as determinization (required in FSA composition) is exponential in the number of states. For this reason, SPAF is appropriate only for applications that can tolerate rather long filter generation times.

Swift [12] focuses on packet filtering updates in strict real-time. The ultimate goal is to add a new filtering rule for a TCP session as soon as its three-way handshake is completed, which is done through a tree-like structure similar to PathFinder. This enables also the use of new x86 *SIMD* instructions to perform multiple checks in parallel.

Ruler [24] is a packet rewriter designed to anonymize traffic traces, which can also be used for packet filtering. It introduces a flexible high-level language for deep packet inspection and rewriting, which is mapped on an extension of the FSA model. Since it is based on automata, Ruler shares a degree of similarity with pFSA and SPAF, but its design goals are sufficiently different to produce noticeably distinct results; furthermore, its source language is not general enough to specify complex filter statements or certain commonly encountered protocol structures, such as IPv6 extension headers.

To the best of our knowledge, SPAF is the only packet filter model that uses a FSA-like approach. If we broaden the area of research, we find that only a handful of publications has proposed extensions to the base FSA formalism that might be similar to ours. **pfsr** [25] is a predicate-augmented finite state recognizer, that aims at simplifying the Finite State Automata used in natural language processing. Even if the authors describe in detail their model extension, providing definitions and algorithms, the scope of the predicate that they introduce is quite different from ours, as it is used only to define arbitrary sets of input symbols. **EFSA** [26] models a fast intrusion detection and prevention system by making use of augmented FSA transitions with arbitrary predicates. The EFSA paper, however, does not describe predicates in detail: e.g, predicate optimization, that is a critical issue in packet filtering, is not mentioned at all. **XFA** [27] is also based on an augmented FSA, but states are associated with a generic executable code for efficient pattern matching, which is not appropriate for optimizing predicates in packet filtering applications.

Other packet filtering technologies such as **FFPF** [14] are not described in detail here, as they aim to solve orthogonal problems, such as how to multiplex incoming packets between different packet filters, but do not offer any improvement to the filtering model itself.

The most common filtering architectures (excluding SPAF) tend to rely on *ad hoc* optimizations, often inspired at compiler-oriented techniques, which are applied on the code that has to be executed. Some of them exploit optimizations to coalesce packet accesses or use hardware-efficient assembly instructions. However, no guarantees of optimality can be given; furthermore, many of those optimization algorithms scale exponentially with the number of instructions of the generated filter, which becomes a major problem when the size of the filter grows because of more complex conditions or uncommon encapsulations, including tunneling. Instead, pFSA defines a packet filtering model based on the FSA formalism that guarantees optimal filtering construction (by minimizing the number of

checks on the packet) and that overcomes the SPAF limitation in terms of compilation time. This is due to the capability to derive the FSA from the protocol abstraction, while SPAF adopts a byte-stream approach, that generates a much more verbose automaton compared to our pFSA.

2.3 Finite State Automata with Predicates

This section presents the **pFSA** model, an FSA extension in which transitions are associated with Boolean predicates. The advantage is that a well-defined algebra already exists for FSA, allowing their optimal composition (union, intersection, negation). In Section 2.4 we will present how the pFSA model can be used for packet filtering.

2.3.1 Definition of pFSA

A **Finite State Automaton with Predicates** (or, briefly, **pFSA**) is a "five-tuple"

$$A_{pfsa} = (Q,\ \Sigma,\ \delta_p,\ q_o,\ F)$$

where:

Q is a finite set of *states*;

Σ is the set of *input symbols*;

$\delta_{\mathbf{p}}$ is a *transition function with predicates* (described below);

$\mathbf{q_0}$ is the *starting state*, among those in Q;

F is a set of *accepting states*, among those in Q.

A **transition function with predicates** mimics the meaning of "classic" transitions, but adds the possibility to tune the transition behavior according to a set of **Boolean predicates**, whose semantic is orthogonal to the input symbols of the automaton. It is defined as:

$$\delta_p(q_1,\ \sigma,\ p) = q_2$$

where:

$\mathbf{q_1}$ is the state from which the transition takes place;

σ is the input symbol that triggers the transition, or the special value ϵ (*epsilon*) if no input symbol should be consumed;

p is a Boolean predicate that "activates" the transition, that is allowed to fire only if the predicate is *true*;

$\mathbf{q_2}$ is the state reached by the current transition.

A transition with predicates is called a **p-transition**; if **p** is always *true* (a tautology), the transition is in fact equivalent to a "classic" one.

Figure 2.1 depicts an example of a very simple pFSA, where p-transitions are labeled with the input symbol[1] and the predicate associated with it, in the form `symbol/predicate`. In the example, a p-transition leaves from state `Q1` and reaches state `Q2` for input symbol `a` and predicate `p1`.

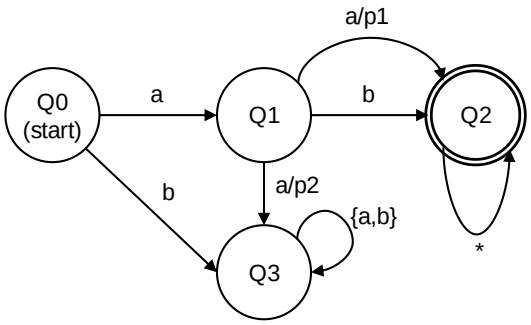


Figure 2.1. pFSA example.

Predicates are part of an arbitrary set of hypotheses and can assume either the *true* or *false* Boolean value. From the pFSA model point of view, each predicate is a "black box" outside the scope of the model, whose actual value cannot be determined *a priori*. In fact, pFSA relies on an external "predicate evaluator" module that will be invoked at *run-time* in order to determine the value of the predicate itself.

We do not pose any particular limitation to the predicates; however, the *predicate evaluator* cannot change the internal state of the automaton, such as move the current state from `Qn` to `Qm`, or change the input string, etc. In other words, the predicate evaluation step must have no side effects other than returning the current Boolean value of the predicate: it is duty of the automaton itself to interpret the returned value and act accordingly. Given this limitation, a predicate could be as simple as *"is the value of variable $x odd?"*, but even a question like *"is IP multicast enabled on the `eth0` network interface?"* is valid, as long as it is possible to give a *true/false* answer. Predicate values are allowed to change only when a new input symbol is consumed; in other words, they are frozen when an ϵ-transition is going to fire. This restriction is needed for the general algorithms to work, but does not have any impact on our usage of the model. More details will be given in Section 2.4.3, after we describe how we use the model to filter network packets, together with some examples (Figures 2.7 and 2.8).

[1]Some transitions (e.g., the self-loop on `Q2`) may be associated with a *star*, which is a compact notation used to include any input symbol that is not handled by other transitions exiting from the same state.

2.3.2 Running a pFSA

The state machine defined by pFSA looks similar to the one of a "classic" FSA. Execution starts with the automaton in the *start state*. As long as an input symbol is available, the automaton reads it and follows any available transition exiting from the current state and labeled with that symbol; the landing state becomes the next current state. Non-determinism is allowed in pFSA, and ϵ-transitions (that do not consume any input symbol) are permitted.

Whenever, according to the current state and the input symbol received, a p-transition should be activated, its predicate is inspected and its current Boolean value is returned; that transition fires if the predicate is found to be *true*, otherwise another transition is taken. Note that if multiple p-transitions have the same start state and are labeled with the same input symbol, a subset of them (from zero to all) might fire at the same time, according to the values taken at run-time by their predicates: this is an important issue to consider when stating whether a pFSA is deterministic or not (more details in Section 2.3.3). Referring to Figure 2.1, if the symbol `a` is received when the control is in state `Q1`, two p-transitions might fire: the one that leads to state `Q2` can fire depending on the value of predicate `p1`, while the one that leads to state `Q3` can fire according to the value of predicate `p2`. Only the run-time values of `p1` and `p2` can clarify which state(s) will be reached: either `Q2`, or `Q3`, or both of them or none.

2.3.3 Determinism

Determinism is important for multiple reasons. First, the FSA complementation algorithm requires the input FSA to be deterministic. Second, complementation is needed also for the intersection algorithm, if the latter is implemented using first De Morgan's law ($A \cap B = \overline{\overline{A} \cup \overline{B}}$). Third, deterministic automata are much easier to translate into machine code: only one state is active at any instant, hence backtracking is not required. A non-deterministic machine, instead, may have to "guess" which path to follow; that is, the algorithm might have to try all the possible routes to the solution, therefore increasing computation times on strictly sequential machines.

A pFSA is **deterministic** if it does not include any ϵ-transition and, for each state, for each input symbol and for all possible values of the Boolean predicates, there is exactly one enabled, outgoing transition.

While this definition looks simple, stating whether a pFSA is deterministic or not may be complicated in practice, because the outcome depends on the values of the predicates, that can be evaluated only at run-time. For instance, if two transitions labeled with `p1` and `p2` exit from the same state and are associated with the same input symbol (such as in Figure 2.1), that pFSA is possibly non-deterministic, as both transitions might be enabled at the same time. Conversely, if those transitions are labeled with `p1` and $\overline{\texttt{p1}}$ there is no determinism issue, as the logic rules assert that exactly one between those predicates is true at any instant. Consequently, if in a given pFSA no state has multiple transitions with the same symbol (i.e., the base FSA is deterministic) and the predicates associated with the transitions exiting from every state, labeled with the same input symbols, are

only in the form `p1` and $\overline{\texttt{p1}}$, that pFSA is still deterministic.

2.3.4 Algorithms

One of the main advantages of reusing the FSA formalism is that many definitions, algorithms and optimizations from the literature (e.g. [21]) can be reused with little effort.

For example, **union** and **complementation** algorithms require no changes. The first algorithm merges two automata by adding a new starting state and connecting it to the starting states of the two original automata with a couple of ϵ-transitions; hence, predicates are not considered at all. The second algorithm requires only to flip the accepting status of all states, provided that the input pFSA is deterministic; hence, again, predicates do not make any difference. No extra effort is required for the **intersection** algorithm, as it can be easily implemented on top of union and complementation by using first De Morgan's law.

These algorithms, however, may produce pFSA that are possibly not deterministic and/or redundant, hence requiring additional procedures (such as **determinization** and **minimization**[2]) in order to produce better automata. Unfortunately, these algorithms cannot be plainly reused for pFSA.

Before presenting the determinization algorithm in detail, we will give a brief look at the main ideas behind the procedure: *(i)* predicates Cartesian product and *(ii)* predicate anticipation. Both of these procedures will be used later, in the determinization algorithm.

Predicates Cartesian product: It is used to determinize a pFSA in which a state has multiple outgoing transitions, all triggered by the same input symbol but associated with different predicates[3], such as in the leftmost part of the fully specified pFSA in Figure 2.2. To guarantee the determinism property, the pFSA is determinized by introducing a number of transitions that is equal to the Cartesian product of the existing predicates; refer to the central part of Figure 2.2, where each transition is terminated on the state that would be activated in the original pFSA, or on a new state (e.g. `Q12`) that captures multiple states of the original automaton. This way we can guarantee that only one out of the four transitions leaving from `Q0` for input symbol `a` can be true, independently from the actual values of predicates `p1` and `p2` at runtime. The resulting pFSA can be further optimized by additional algorithms, such as compaction of indistinguishable states: e.g., in the automaton in the rightmost part of Figure 2.2, state `Q12` has been merged with `Q1`.

Predicate anticipation: Sometimes, the pFSA determinization algorithm may move a predicate bound to an ϵ-transition over another transition that precedes it, with some additional adjustments; this is useful when the preceding transition is not already associated with a predicate, and the anticipation allows a state simplification. This transformation is possible because it is guaranteed that the predicate has the same Boolean value in

[2]Even if determinization and minimization are two distinct algorithms, the latter is usually executed immediately after the former; therefore, they are often presented as being part of the same procedure.

[3]Starting from Figure 2.2, the notation `!predicate` (borrowed from popular programming languages) is used to express a negated condition.

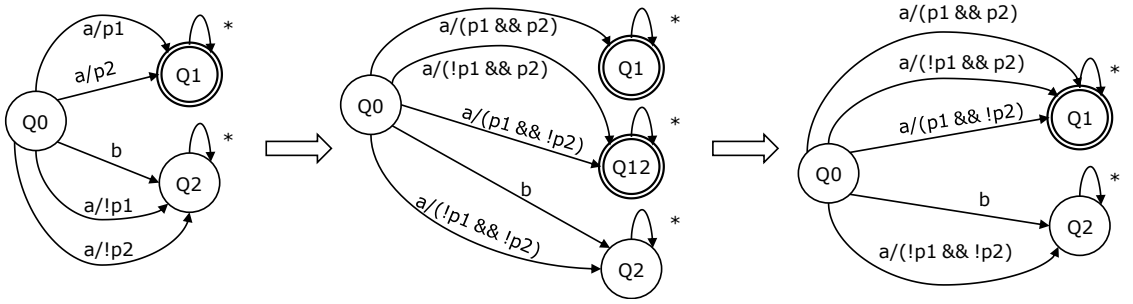


Figure 2.2. pFSA predicates Cartesian product.

both cases: as already outlined in Section 2.3.1, predicate values are allowed to change only when a new input symbol is consumed. A simple example is shown in Figure 2.3, where predicate `p1` on the ϵ-transition between states `Q2` and `Q3` is moved over the previous transition from `Q1` to `Q2`. However, moving the predicate requires the creation of two transitions (one labeled as `a/p1`, the other as `a/`$\overline{\texttt{p1}}$) and the duplication of state `Q2` (second step of Figure 2.3). The final pFSA, obtained by removing the ϵ-transition and by compacting the states that are indistinguishable (i.e., `Q2` and `Q3`, and `Q2'` and `Q4`), is depicted in the rightmost part of Figure 2.3.

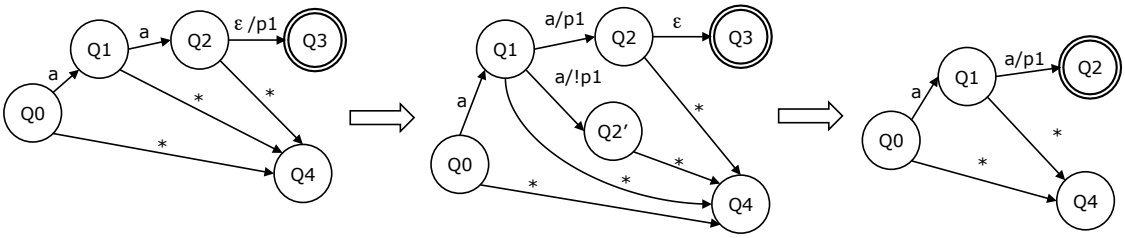


Figure 2.3. pFSA predicates anticipation.

We will now discuss the determinization and minimization algorithm in more detail, listing how predicates are considered: *(i)* when trying to determine the states reached from the current state upon the receipt of a given input symbol, *(ii)* when calculating an ϵ-closure and *(iii)* when determining if two states are indistinguishable by testing the output of function δ_p (i.e., they have exactly the same output transitions that bring exactly to the same output states).

When **computing the reachable set**, the transitions exiting from any given state s are considered, for each input symbol σ. If multiple transitions exist with the same symbol that potentially can fire because one or more predicates are present, their Cartesian product is computed and all possible landing states are evaluated: Boolean rules ensure that only one of the transitions out of the product can fire at any given instant in time.

The **ϵ-closure** of a set of states recursively adds to the set of states S all those that

are reachable, through an ϵ-transition, from any state already in S. If some predicates are found on those ϵ-transitions, their Cartesian product is computed, possibly also against the predicates already discovered in the previous step.

Finally, the modified **transition function δ_p** is used in the state compaction step to detect if two states are equivalent and can be merged together. To achieve this, the transition function δ_p checks whether, upon the receipt of a given input symbol σ, two transitions exist that lead from the couple of states under testing to any other couple of states that were already found distinguishable. In a pFSA, if a predicate is present over a transition, then, for the distinguishability test, all the other transitions that are associated with the same input symbol and exiting from the same state must be considered as well.

2.3.5 Predicates composition

The algorithms presented in the previous section are used to combine together more pFSA, leading to a new, equivalent pFSA that retains all the properties guaranteed by the pFSA formalism. The example in Figure 2.4 shows the union of two simple pFSA through the required processing steps: a new state is connected to the original pFSA through two ϵ-transitions (in the middle), then the final pFSA that comes after determinization and minimization is shown at the right. The example in Figure 2.5 looks more complicated as it shows the intersection between two pFSA, which occurs by transforming that operation into a set of union and negation steps[4].

It is evident from the examples how the pFSA determinization algorithm analyzes all possible combinations of the Boolean values of the predicates. This may become a problem in case of complex pFSA, e.g. obtained by merging several simpler pFSA together, as the number of possible combinations may grow exponentially. This represents a non negligible challenge when the pFSA has to be actually translated into executable code, because of the large number of expressions that have to be evaluated at runtime. We feel that different use cases might benefit from different predicate optimizations; given that our application domain focuses on packet filtering, we will present in Section 2.5 how we deal with the predicate composition in that scenario, by means of a predicate optimization formalism called *protoFSA*.

2.4 pFSA for packet filtering

Although the pFSA model is rather general and can be adapted to different contexts, this chapter focuses on its application in packet filtering and shows how multiple filters can be combined together with a solid guarantee of optimality in terms of number of checks on the packet. This section focuses on this objective, presenting how the pFSA model can be used to describe a generic filter, exploiting pFSA properties to reduce (and optimize) complex filtering expressions. To do so, we should be able to translate a filtering expression into

[4]In Figures 2.4 and 2.5, transitions with dashed lines are redundant and may be deleted, as they are included in the default `'*'` arc.

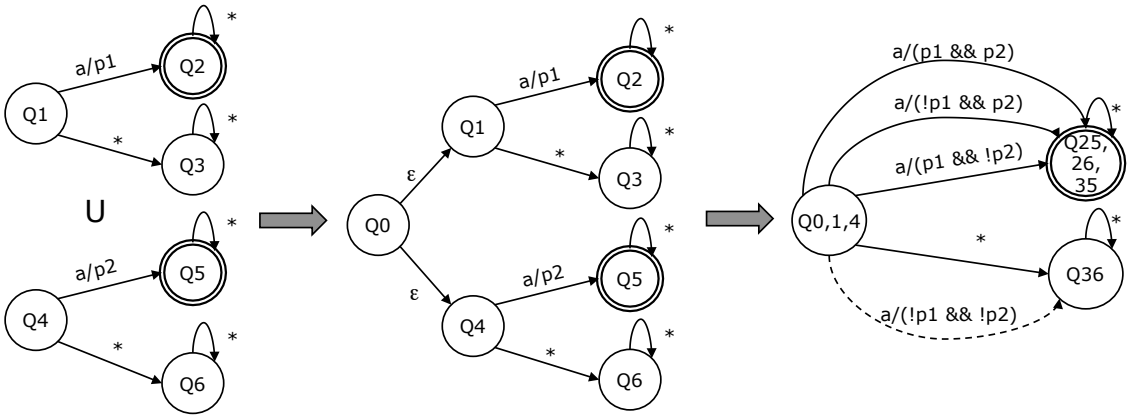


Figure 2.4. Example of a pFSA with complex predicates: the *union* case.

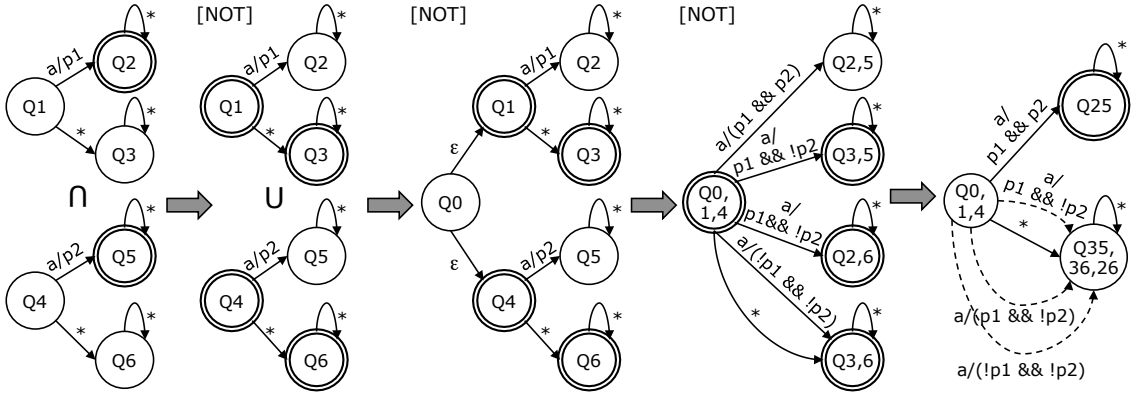


Figure 2.5. Example of a pFSA with complex predicates: the *intersection* case.

an equivalent pFSA, so that: *(i)* if a packet matching the provided filter is given to our system, the pFSA should end in an accepting state; *(ii)* otherwise, if the packet does not match, the pFSA should end in a non-accepting state.

We will describe how the packet filtering machinery is mapped in the pFSA model: namely, how states, symbols and predicates are defined. An overview of the system is given in Figure 2.6, while a detailed view of each block will be given in Section 2.4.5.

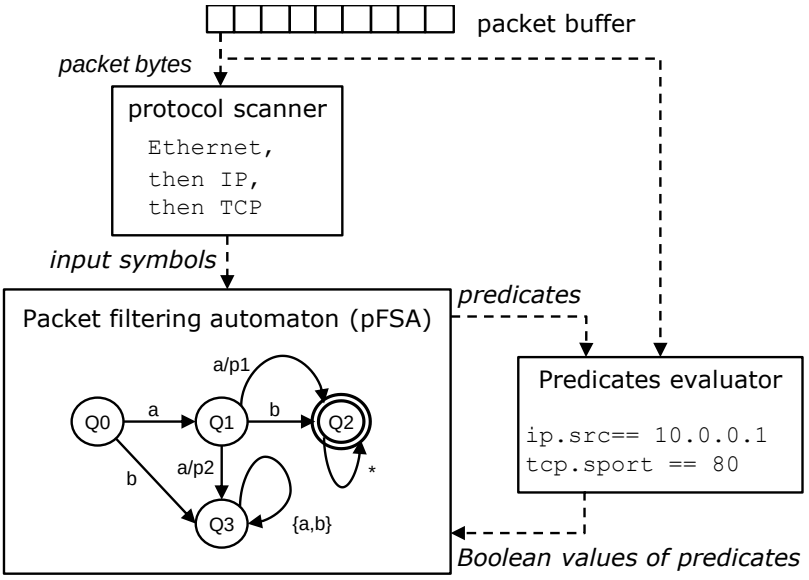


Figure 2.6. Overview of the system in which pFSA are used for packet filtering.

2.4.1 States

The initial construction of the pFSA associates each state with a network protocol, that represents the protocol that has been reached while scanning the current packet[5]. For instance, when the `ip` state becomes active, it means that the `IP` protocol has been found in the current packet and that the protocol scanner is going to read the first byte associated with it. As a consequence, for simple pFSA, the set of accepting states includes only those states that match the protocol requested by the filter: for instance, in the pFSA modeling the filter that selects only `ip` traffic (e.g., in Figure 2.7), the state labeled `ip` would be the only accepting state. More details about the important bonding between states and network protocols will be presented in Section 2.4.4.

2.4.2 Input symbols

In a pFSA for packet filtering, each input symbol represents a single encapsulation rule, i.e., a sort of "jump" from a protocol to the next. For instance, the symbol `ethernet-to-ip` is associated with a transition that goes from a state that represents the `Ethernet` protocol to another that represents `IP`, as shown in Figure 2.7. If the pFSA receives this symbol at runtime, it means that the packet currently under examination contains an instance of

[5]This rule does not apply to the starting state, which represents the state of the automaton before the packet scan has started.

`IP` directly encapsulated inside an `Ethernet` header[6].

In our system, input symbols are generated by a separate module (the **protocol scanner** of Figure 2.6), which inspects the incoming packet, analyzes the protocols in it, generates the symbols and passes them to the pFSA engine; for each encapsulation found in the current packet, a new input symbol is generated. For instance, if a packet contains, in this order, the `Ethernet`, `IP` and `TCP` headers, then three input symbols are passed to the pFSA: `begin-to-ethernet`, `ethernet-to-ip` and `ip-to-tcp`.

The input symbols that the pFSA expects to receive (the $\mathbf{\Sigma}$ alphabet) are derived from a *protocol database*, that is provided to the engine that builds the pFSA at filter compilation time. More details about the protocol database and the building process are in Section 2.4.5.

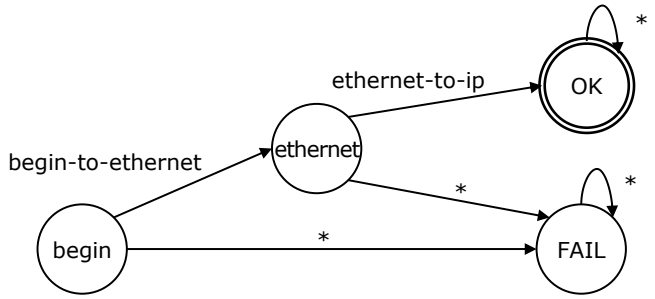


Figure 2.7. Example of a pFSA for the filter `ip`.

2.4.3 Predicates

While filtering packets, predicates are modeled as hypotheses on specific "properties" of the protocols included in the packet itself, possibly combined together with Boolean operators. In our work, predicates are expressed with a "basic block" in the form `<protocol_field> <operator> <value>` Currently, basic comparison operators are supported ($\leq$, $<$, $=$, $\neq$, $>$, $\geq$) and `value` must be a constant. Obviously, filtering conditions can become more complex when multiple predicates are combined together with the classical Boolean operators (`and` , `or` , `not`).

Figure 2.8(a) shows a simple pFSA for the filter `ip.src == 1.1.1.1`. Note that, differently from Figure 2.7, the `ip` state is no longer accepting: the `ip` state is connected to the actual accepting state through an ϵ-transition with the `ip.src == 1.1.1.1` predicate. If this predicate is found to be *false* at run-time (because the IP source address does not match the value `1.1.1.1`), then the path towards the accepting state is effectively

[6]The only exception to this rule applies to the input symbols that represent the first protocol of each packet: since there is no explicit "previous protocol", the fictitious `begin` protocol associated with the starting state is used, and the link-layer associated with the parsed packet determines the input symbol for the first transition.

barred, therefore rejecting the packet. Figure 2.8(b) shows the same pFSA after running the predicate anticipation algorithm, which transforms the pFSA into a deterministic automaton. It is worth noting that the two forms (a) and (b) of the given pFSA are completely equivalent and it is possible to transform one into the other, if needed.

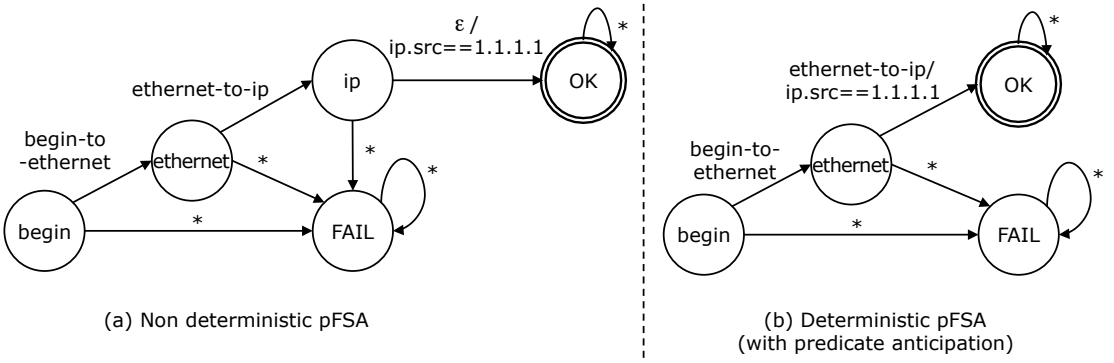


Figure 2.8. Example of a pFSA for the filter `ip.src == 1.1.1.1`.

The actual Boolean value of the predicates is evaluated by a dedicated module (the **predicates evaluator** of Figure 2.6), that is logically separated from the protocol scanner. Whenever the pFSA encounters a predicate at runtime, the evaluator is invoked and the current Boolean value of that predicate is returned.

It is worth remembering that predicates can be evaluated only when the corresponding transition is about to fire and cannot be precomputed, because their value might change every time a new input symbol is consumed. For instance, the Boolean evaluation of `ip.src == 1.1.1.1` may result in different values when filtering a packet that contains a `ip-in-ip` tunnel, depending on whether we are operating on the inner or outer `IP` header. This case is not handled in Figure 2.8 to keep the example simpler.

2.4.4 States and network protocols

Due to the properties of the pFSA model applied to packet filtering, each state can be associated with a precise network protocol. This is needed at a later stage in order to translate each state into filtering code (e.g., assembly instructions) and to be able to perform predicate optimizations, as presented in Section 2.5.1. This relation is maintained also after merging and optimizing multiple pFSA, when multiple states are joined together.

In fact, we can envision three cases in which multiple states are merged. The first case occurs when computing the ϵ-closure, which happens only when a new state, not associated with any protocol, is added in front of the two FSA. This requires to merge together both (semantically identical) `begin` states of the original automata. The second case occurs when two states are found to be equivalent, i.e., they have the same set of outgoing transitions. As transitions are associated with a specific protocol encapsulation rule, having the same set of transitions means that the states that are going to be merged

refer to the same protocol. The third case refers to final states, which are merged independently from the protocol they are associated with; however, the association with the originating protocol is useless in this case, because the automaton is going to terminate anyway.

This nice property of strong relation between states and protocols can be apparently lost when some optimizations (particularly, predicate anticipation) come into play. For instance, Figure 2.10 presents an example in which an intermediate state is associated with the reachability of a given protocol field, i.e., the pFSA reaches field `ip.src`. However, this does not represent a problem as input symbols (which represent protocol encapsulation rules) guarantee that two states can be merged only if they are reached through the same encapsulation rule.

Also note that multiple states, associated with the same protocol, can coexist. Particularly, this may happen in two cases: *(i)* when the same protocol is present on multiple (disjoint) paths from the beginning state to any final state, as the pFSA can define different independent paths that cross the same protocol, and *(ii)* when the same protocol is present on the same path, but it refers to different instances, e.g., the inner and outer `IP` headers of an `ip-in-ip` encapsulation.

2.4.5 Building a pFSA for packet filtering

The process that creates the pFSA that represents a given packet filter involves different components, as presented in Figure 2.9. The packet filtering pFSA is the result of the combination of the **filtering string**, which represents the actual filtering statement, and a **protocol database**, which features a description of the protocols in terms of fields and encapsulation rules (although in this step only the latter is considered). Encapsulation rules specify how protocols are encapsulated one into the other, resulting into a directed, potentially cyclic, graph. For instance, there will be an entry that states that the `IP` protocol can be found inside `Ethernet`, but there will be no entry for `TCP` inside `Ethernet`. The protocol database should also mark the protocols that can be found at the beginning of a packet (i.e., link-layers such as `Ethernet` or `WiFi`), in order to highlight which protocols represent some sort of "starting nodes" of the encapsulation graph; in our implementation those link-layer protocols follow a fictitious "begin" protocol. The pFSA model is agnostic with respect to the protocol database, as long as it includes the required information; in fact, the choice of this external component is under the responsibility of the specific pFSA implementation.

The first step towards the pFSA creation is parsing the *filtering string* itself, splitting it in basic tokens, i.e., statements that express a condition operating on a single protocol or a protocol field, chained together with Boolean operators. A distinct pFSA is generated for each of these blocks, which are combined together using the algorithms presented in Section 2.3.4, therefore obtaining the final pFSA. As each portion of the tokenized filtering string refers to a single protocol, it is used to traverse the encapsulation graph and to select all the paths that connect the *starting protocol* (that represents the starting state of the automaton) to that protocol. For instance, all paths that result useless for the given filter are discarded in this step. All nodes and edges selected are then used to build the

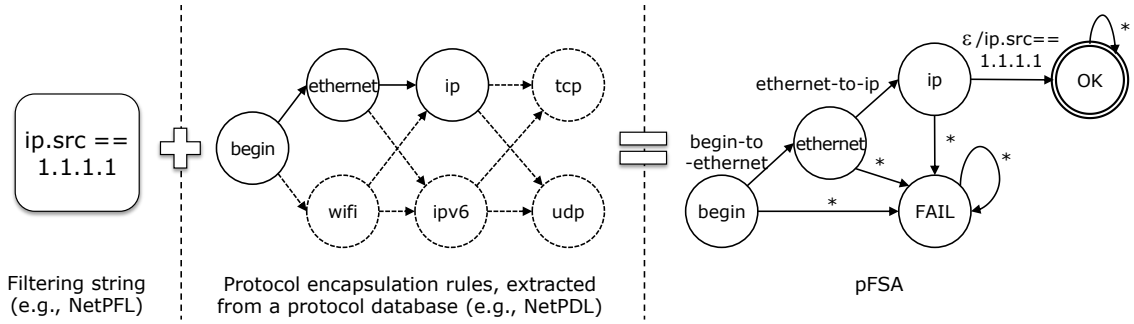


Figure 2.9. Building steps for a simple pFSA.

pFSA, transforming each encapsulation into a possible input symbol for the automaton. An additional state is created, representing the non-accepting condition (i.e., when the packet does not match the filter); every other state is then connected to this failure state using a *star* transition[7].

In the end, an accepting state must be specified. If the filter statement does not include conditions on protocol fields (e.g., `ip`), then the pFSA state associated with that protocol is marked as accepting. Otherwise, the state representing the above protocol is connected to a newly created accepting state by means of an ϵ-transition, labeled with the provided predicate. Finally, a looping transition that fires for all symbols is added to each accepting state, to ensure that the resulting pFSA is completely specified.

The examples shown in Figure 2.7 and 2.8 were created with this algorithm: in both cases the `begin` state corresponds to the *starting protocol* in the above description. Those examples show also that the FSA creation process can lead to non-deterministic pFSA, such as in Figure 2.8.

Figure 2.10 represents a more complex example: a pFSA already determinized for filter `ip.src == 1.1.1.1 and tcp.dport == 80`. The filter appears optimal in the number of tests: only one path leads to state `OK`, which includes the verification of both conditions present in the filter. If the first test fails, the failure state is reached immediately, ignoring the run-time value of the second predicate.

2.5 Predicates optimization

The optimization of filtering predicates represents a critical issue in order to effectively model packet filters; in particular, some applications may be extremely sensible to the problem of predicate composition previously introduced in Section 2.3.5. In fact, a model that guarantees optimality with respect to protocol encapsulations is still not enough for

[7]It is worth remembering that the "star" represents a compact notation that replaces all the input symbols (and predicates) that are not used by the other transitions exiting from the current state.

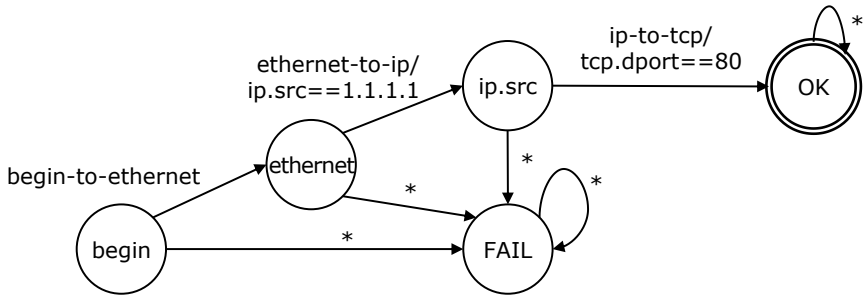


Figure 2.10. Example of a deterministic pFSA for the filter `ip.src == 1.1.1.1 and tcp.dport == 80`.

those applications that require complex filtering expressions operating on protocol fields: these are somewhat "outside" the pFSA model and hence, so far, are not optimized at all.

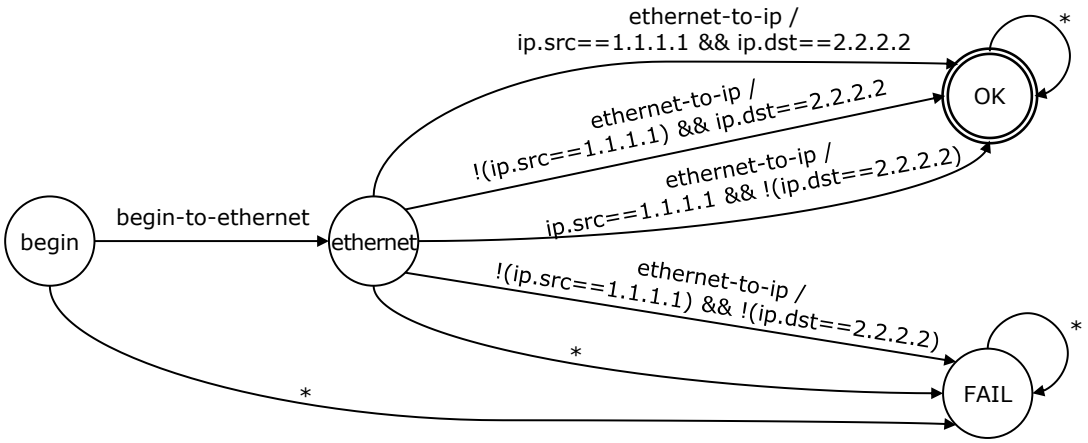


Figure 2.11. Example of a deterministic pFSA for the filter `ip.src == 1.1.1.1 or ip.dst == 2.2.2.2`.

For instance, Figure 2.11 represents a deterministic pFSA modeling the filter `ip.src == 1.1.1.1 or ip.dst == 2.2.2.2`, which is then translated into a pFSA that requires the analysis of four predicates when the `ethernet-to-ip` input symbol is received. This exponential explosion in the number of transitions might be troublesome in complex (but very common) packet filters, e.g., those that account hundreds of tests over the same protocol fields, such as Access Control Lists operating on IP addresses. All those transitions must be evaluated by the predicates evaluator in order to determine which one will fire (if any), hence posing a substantial run-time overhead when trying to resolve the current Boolean value of multiple, arbitrarily complex predicates.

However, note that, due to the potential Cartesian product on filtering predicates, the predicate evaluator is called several times for similar expressions. For example, it is evident that the Boolean values for predicates `(p1 && p2)` and `(p1 && !p2)` are correlated and

some optimizations are possible.

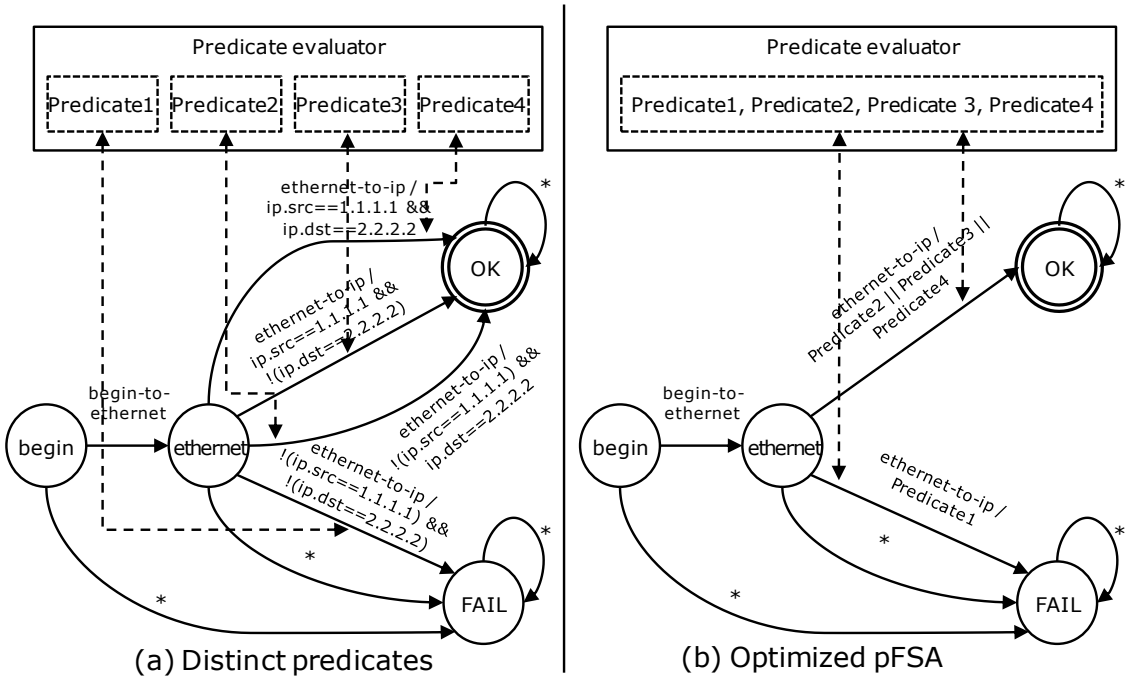


Figure 2.12. The main idea behind the "multilevel" implementation feature. Predicates are merged within the same predicate evaluation block, leading to a simplification of the base pFSA: multiple transitions are merged together, paving the way for further optimizations at the predicate level.

2.5.1 Overview

We optimize the behavior of the predicate evaluator by operating in three steps: *(i)* we merge multiple predicates together, enabling the evaluation of multiple queries in a single pass; *(ii)* we simplify the pFSA by compacting the transitions that result redundant when looking at the base automaton (e.g., because multiple transitions land on the same state), and *(iii)* we analyze the semantic of the predicates looking for possible optimizations at compile time, enabling a faster predicate evaluation step at run-time (e.g., `tcp.sport == 80 && tcp.sport > 1024` is always `false`).

For the first step we created a block that can merge multiple queries coming from different transitions, instead of having different predicate evaluators for each expression such as in Figure 2.12(a), which is possible because the pFSA model does not mandate the internal architecture of the predicate evaluator. If predicates operate on the same protocol field (which is rather common), their evaluation is potentially faster.

The second step (shown in Figure 2.12(b)) simplifies the layout of the pFSA when possible. For instance, the three transitions between states `ethernet` and `OK` can be

compacted into one, associated with the logical `or` of the three predicates, thus enabling further optimizations in the next step.

The third step minimizes the operations needed to evaluate the expressions by restructuring the internals of the predicate evaluator. For instance, given the predicates in Figure 2.12, we can structure the predicate evaluator so that the condition `ip.src == 1.1.1.1` is checked once and then its result is reused for all expressions; or, the test on `ip.dst` is not performed if its value does not change the final result.

The effectiveness of the predicate optimization presented above is a direct consequence of the property that associates pFSA states with a given instance of a network protocol, presented in Section 2.4.4. By construction, all predicates operating on a given instance of a protocol will be associated with transitions exiting from the same pFSA state, therefore becoming part of the same Cartesian product and enabling the predicate evaluator to optimize them all at once.

2.5.2 Going multilevel: the protoFSA

In order to effectively optimize predicates, we need to *(i)* define a model for filtering predicates that is able to efficiently merge filtering predicates when combining different pFSA, guaranteeing optimality with respect to the number of checks done on the protocol fields, and *(ii)* efficiently map filtering predicates to the chosen model.

Our idea is to create another set of FSA that sits on top of the pFSA and is in charge of the optimization of the predicates that result from the same Cartesian product, i.e., that are associated with a set of transitions exiting from the same pFSA state. Each of those new FSA is called **protoFSA**, because it is associated with a given instance of a network protocol. While the pFSA is the base model that handles the entire packet filter, each protoFSA is in charge of the optimizations performed among all the predicates on transitions exiting from a given pFSA state.

Formally, for each state q_i in the pFSA that has a number of outgoing transitions originated by the same Cartesian product Π_i, we define (for each Π_i) another Finite State Automaton:

$$A_{protofsa} = (S,\ \Sigma,\ \delta,\ s_o,\ F)$$

dedicated to predicates optimization, where:

S is a finite set of *states*, associated with the protocol fields referenced by the predicates;

Σ is the set of *input symbols*, which consists in the union of the set of values syntactically valid for each protocol field (e.g., a predicate operating on an IP address and on the IP TOS byte originates $2^{32} + 2^8$ possible input symbols);

δ is the *transition function*, which takes into account whether a condition on a specific field triggers the analysis of a subsequent condition on another field;

s_0 is the *starting state*;

F is a set of *final states*, whose cardinality is equal to the number of outgoing transitions involved in the Cartesian product Π_i.

A protoFSA is still a FSA and consequently inherits all the properties guaranteed by that formalism (e.g., composition, optimality). Each protoFSA can be either deterministic or non-deterministic; however, in our implementation, for simplicity and efficiency, we transform those structures into a deterministic automaton before the final translation, i.e., when the model is converted into running code.

2.5.3 Building a protoFSA

If we analyze all complex predicates originated by the same Cartesian product Π_i, it can be formally proven that the following two properties hold: *(i)* all predicates are in the form of basic blocks (`<protocol field> <operator> <value>`), joined together in logical `and`; *(ii)* the predicates include exactly the same number of basic blocks, operating exactly on the same protocol fields, all referring to the same protocol. Because of property *(i)* and since the commutative property holds for the Boolean `and` operator, we can rewrite the entire predicate string so that basic blocks will be *strictly ordered*, based on the protocol field they refer to[8].

To build a better protoFSA out of each Cartesian product, it would be preferable if, at creation time, we could identify explicitly (but not necessarily enumerate) the set of values that satisfy the condition of each predicate. However, since each basic block compares the protocol field against a *constant* value, this property automatically holds.

Each basic block is translated into a minimal FSA in which the protocol field is associated with a state, while the space of its possible values is used to define the transitions to the `OK` and `FAIL` states[9]. If, based on the protocol fields ordering mentioned above, a basic block refers to a field other than the first one, a set of states referring to its "preceding" fields is pre-pended to the state associated with the state itself. In other words, state s_i, associated with predicate P_i, is preceded by states $s_1 \dots s_{i-1}$, associated with predicates $P_1 \dots P_{i-1}$. Preceding states selected this way are connected with a default transition, such as in the second basic block (bottom left) in Figure 2.13. The `OK` and `FAIL` states are then associated with the transitions (in the base pFSA) that originate the current query to the predicate level.

The next step consists in building the whole protoFSA, by merging together all predicates that result from the same Cartesian product. The final protoFSA has as many final states as the number of predicates resulting from the Cartesian product, each own mapped to a p-transition of the base pFSA. However, multiple p-transitions in the pFSA can be merged together if their ending states are not distinguishable (in terms of the minimization algorithm), so we have a chance to optimize again the protoFSA through the well-known FSA composition and optimization algorithms. For instance, in Figure 2.12(a), predicates

[8]The chosen evaluation order does not matter (e.g., alphabetic comparison among protocol field names, or the order in which those fields appear in the packet), as long as it is kept consistent.

[9]Although formally each state should include a distinct transition for all the possible input symbols, in our protoFSA building process we take into account that some symbols cannot be received when in a given state (e.g., the symbols related to the IP TOS byte cannot be received when examining an IP address), hence simplifying the translation of the protoFSA structure in running code.

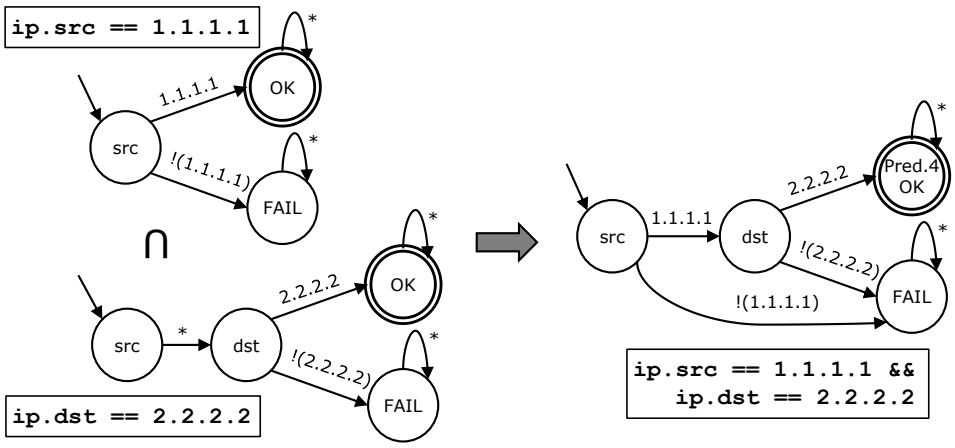


Figure 2.13. Example of composition of the predicate `ip.src == 1.1.1.1 and ip.dst == 2.2.2.2`, corresponding to predicate P_4 in Figure 2.12.

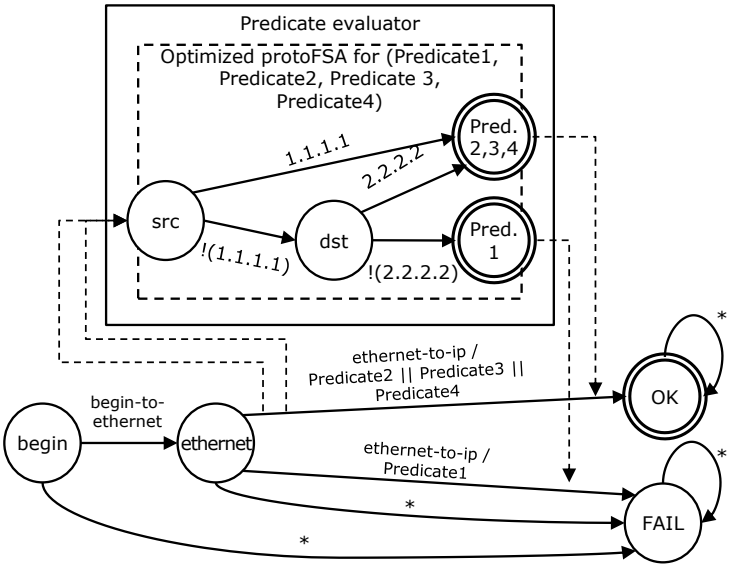


Figure 2.14. Example of the protoFSA created in Figure 2.12, composing predicates P_1, P_2, P_3 and P_4, and the resulting optimized protoFSA.

P_2, P_3 and P_4 lead to the same state; consequently the protoFSA can be further optimized, resulting in the final form shown in Figure 2.14.

2.5.4 About optimality

We can now explain why the claim of the optimal number of checks on the packet is obtained by construction. When the final pFSA is built, the number of checks needed

to recognize a matching packet is equal to: *(i)* the number of protocol encapsulations, plus *(ii)* the number of checks on protocol fields. *(i)* is optimal because of the way the individual pFSA are created and aggregated together, since the final automaton receives as many input symbols as the number of protocol encapsulations present into the packet. *(ii)* is optimal for a similar reason: by construction, the protoFSA consumes as input the minimum number of symbols needed to resolve the Boolean value of a predicate, hence it is possible to minimize the number of checks on protocol fields.

2.5.5 Predicates and ranges

The protoFSA creation mechanism presented in Section 2.5.3 may lead to an automaton with a huge number of symbols, which may represent a problem when defining the transitions exiting from each state, since their cardinality is equal to the number of symbols. In fact, the explosion in the number of transitions affects both the memory occupancy and the computational complexity of the FSA algorithms. In order to overcome this problem, whenever possible we group symbols into ranges, using the full enumeration of the symbols only when needed (e.g., when ranges become very complex). For instance, if a predicate specifies a precise IP address such as in Figure 2.13, we define only two transitions, one for the path that leads to success (associated with the proper IP address, e.g., {`1.1.1.1`}) and the other for all the remaining symbols (e.g., Σ `-` {`1.1.1.1`}).

2.6 Implementation

The proposed pFSA model has been implemented in the NetBee[10] library, which features an experimental compiler that creates run-time code for the NetVM [15] virtual machine. The front-end compiler [16] takes the filtering expression expressed as a NetPFL [29] string and a NetPDL [28] protocol database to generate an in-memory representation of the pFSA filter. This code is then translated into NetIL code, a NetVM-specific assembly-like language. The generated code can be executed in a NetVM interpreter, or compiled Just-In-Time (JIT) if a backend compiler is available for the target architecture. The pFSA abstraction has been implemented inside the front-end of the aforementioned high-level compiler.

The NetPDL technology, which consists in user-editable XML files, allows us to decouple the protocol database from the code that parses and handles network protocols. For instance, our NetPDL-based implementation of the pFSA can operate on all the protocols supported by the NetPDL language, and NetPDL files can be changed dynamically, without having to recompile the code that generates the pFSA.

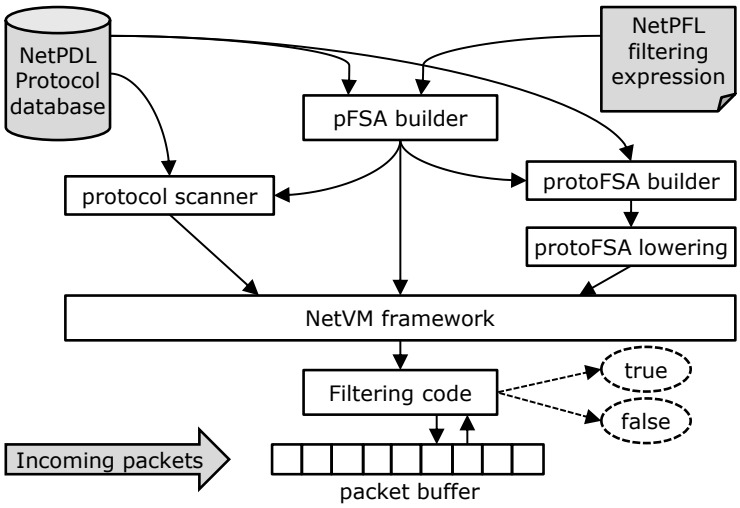


Figure 2.15. Overview of the building blocks in our prototype.

2.6.1 Overview

Figure 2.15 shows an overview of the code generation system implemented by our prototype, which mimics the general architecture presented in Figure 2.6. The *pFSA builder* takes the protocol encapsulation graph (dynamically extracted by the NetPDL protocol database) and the filtering expression and creates the actual pFSA that implements the packet filter. The tokens that allow moving from one pFSA state to another are generated by the *protocol scanner*, which (again) uses the NetPDL protocol database to translate encapsulation rules into running code. Finally, the *protoFSA builder* creates a set of protoFSA, each one dedicated to a single Cartesian product originated by the pFSA. Each protoFSA is then handled by the *protoFSA lowering* module, which takes care of some implementation-dependent optimizations, presented later in Section 2.6.3. All the aforementioned blocks generate the proper data structures according to the primitives exported by the NetVM framework, which finally merges all the code in order to build the actual filtering program.

2.6.2 Protocol scanner

Our FSA-based approach relies on the possibility to generate a sequence of input symbols that correspond to the list of protocols contained at runtime in a given packet. Although the **protocol scanner** is a logically separated module, in our implementation its operations are actually performed by the same assembly program that implements the pFSA related to the given protocol filter. For instance, when generating the NetIL code for

[10] `http://www.nbee.org`

a state, the encapsulation definitions for its protocol are read from a NetPDL database and the corresponding NetIL code is generated and appended to the previously generated code.

2.6.3 Predicate evaluator

The **predicate evaluator** operates in two steps. The first one (**protoFSA builder**) handles each protoFSA generated during the pFSA construction and optimizes its behavior using the well-known FSA algorithms, albeit slightly modified in order to handle transitions based on ranges instead of single values. The second step (**protoFSA lowering**) implements the lowering of the previous high level structure into running code, i.e., a set of proper assembly instructions that implement the protoFSA.

In our implementation both steps are confined into a separate library that takes into account range-based optimizations: all numeric comparisons on a protocol field (involving both range and equality operators) are rearranged into a tree, organized to easily recognize impossible outcomes (e.g., `tcp.dport == 80 and tcp.dport > 1024`). In addition, particular attention has been made in order to lower the code originated by each protoFSA state in the most efficient way. When all transitions exiting from a state include only precise values (such as in the filter `tcp.sport == 80 or tcp.sport == 8080`), the code will be translated into a `switch-case`. When dealing with ranges, instead, the protocol field is initially checked against the bounds of the wider range and, if necessary, against the smaller ones; the comparison for equality against some constants is deferred at the end, if the value is found to lie in the appropriate range. An example can be seen in Figure 2.16.

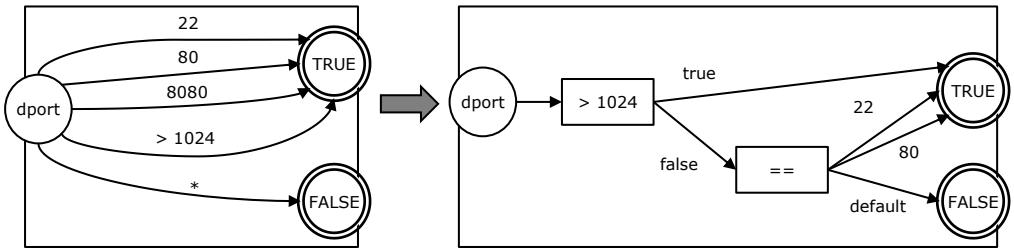


Figure 2.16. Example of a predicate that specifies multiple comparisons against the same protocol field, generated with the filter: `tcp.dport > 1024 or tcp.dport == 80 or tcp.dport == 22 or tcp.dport == 8080`. Note the tree structure and the removal of the redundant checks.

2.6.4 Code generation

Even if the pFSA formalism and the companion protoFSA components are able to create FSA that guarantee the minimum number of checks on the packets for any given packet filter, the code generation process is not guaranteed to maintain this property. In fact, although the final filtering code is created at the best of our knowledge, we cannot formally

prove that it enables each packet to be processed with the smallest number of checks on protocols and fields. However, from the practical point of view, the characteristics of the NetVM framework allow us to speculate that, if the generated code is not optimal, it is very close to it. For instance, the NetVM framework implements many data-flow and control-flow optimizations (more details in [15]) and our experimental evaluation proves that our speculation is correct in case of the most common filtering expressions, while in other more complex cases the code is rather close to optimality.

For example, a class of non-optimal filters originates from the fact that the protocol scanner and the protoFSA builder operate independently, hence a filter such as `ip and ethertype==0x86DD` is considered valid. However, the optimization algorithms implemented in the NetVM framework later detect that this is an always *false* filter, as the `ethernet-to-ip` encapsulation requires the `ethertype` field to be equal to `0x0800`. The (missing) early detection of this problem is a limitation of our current approach, which should be addressed in our future work.

2.6.5 Safety

Safety is one of the key problems to deal with when creating efficient packet filters, particularly when JIT techniques are used for code generation. Safety means guaranteeing that the program always terminates (no infinite loops can occur), and that all memory accesses refer to valid offsets.

The strong relationship between pFSA and Finite State Automata should, in principle, help us in guaranteeing that some properties are satisfied ahead of time. For instance, the termination property holds if the FSA keeps consuming input symbols, i.e., reading new bytes from the input packet at always increasing offsets, which (sooner or later) exhausts the input buffer, leading the filtering code to come to an end.

In fact, we can guarantee filter termination in pFSA by checking that each new protocol has an header size greater than zero: each time a new protocol is encountered, the offset inside the packet increases, hence reaching the end of the input buffer at some point. Furthermore, we do not observe any termination problem within each protoFSA, as (by construction) loops are not allowed in any protoFSA block.

With respect to bounds checking, we make use of traditional techniques based on offset validation before loading/storing a value from/into memory. Although this technique can be improved, we did not investigate this issue any further and we decided to make use of the naive algorithm already implemented in the NetVM compiler. We expect that a minor performance improvement could be achieved if a more aggressive algorithm is implemented.

2.7 Validation

The pFSA model has been compared with other packet filters from the state of the art, such as Ruler, BPF and SPAF. Some experiments have been carried out only against SPAF, which represents the sole competitor that supports some of our features, such as arbitrary protocol encapsulations; furthermore, it is also based on the FSA formalism.

Three different test categories were set up to evaluate different aspects of our solution: *(i)* compile-time performance, *(ii)* run-time performance and *(iii)* scalability. These tests are largely inspired at those in [13] and were performed in a very similar environment. All tests were performed on a workstation equipped with an Intel E8400 Core 2 Duo dual-core processor with 4 GiB of RAM, running a 64-bit version of Ubuntu Linux 10.04. Time measurements were performed either using the RDTSC assembly instruction or, for reasonably longer periods of time, the `gettimeofday()` UNIX function. Memory footprint measurements were performed by using the GNU `time` command or, where applicable, using the Java VM memory management methods. All test processes were bound to a single processor, with hot disk and processor caches, and the machine was otherwise unloaded.

2.7.1 Filter compilation time

As a first step, we evaluated the compile-time performance of pFSA and SPAF. The set of filters in Table 2.1 was taken as a reference. Two different protocol databases were chosen: the first one is called *core* and includes only definitions for `Ethernet`, `IPv4`, `TCP` and `UDP`, without any recursive encapsulation; the second one is called *full*, includes also definitions for `VLAN`, `ARP`, `PPPoE` and `IPv6` and some recursive encapsulations: `IPv4-in-IPv4`, `IPv4-in-IPv6` and `IPv6-in-IPv4`.

filter 1	`ip`
filter 2	`ip.src == 10.1.1.1`
filter 3	`tcp`
filter 4	`ip.src == 10.1.1.1 and ip.dst == 10.2.2.2 and tcp.sport === 20 and tcp.dport == 30`
filter 5	`ip.src == 10.4.4.4 or ip.src == 10.3.3.3 or ip.src == 10.2.2.2 or ip.src == 10.1.1.1`

Table 2.1. Sample filters

Figure 2.17 portraits, in logarithmic scale, the time needed for pFSA and SPAF to compile the filters above, either when run with the *core* database or with the *full* one. pFSA running times are broken down in actual *compilation time* (the time needed to get to the final automaton and generate the NetIL code from it) and *optimization time* (the time needed for the data-flow and control-flow optimizations to run over the NetIL code). JIT compilation time for pFSA is not displayed; neither the time required to compile the C code generated by SPAF. SPAF computation times are missing for filters 3 to 5 when executed with the *full* database, because we interrupted those tests when their processing time exceeded 24 hours.

Figure 2.17 shows that the pFSA filter compilation process is several orders of magnitude faster than SPAF, even if we include the optimization time (which is not formally part of the model). The reason can be found in the greater efficiency of the building process of the automaton, which is due to the choice to consider protocols and fields when building the FSA instead of relying on unlabeled bytes in the packet, generating a far

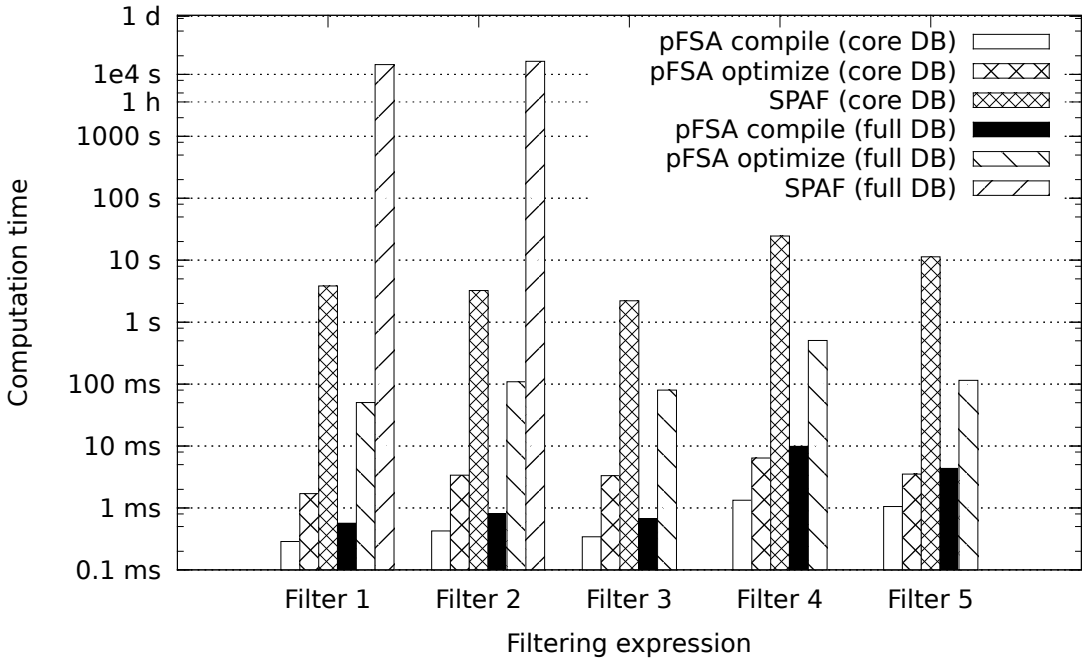


Figure 2.17. Comparison of the time needed by pFSA and SPAF to compile and optimize a filter.

smaller number of states. This has a huge impact on the overall building process, as the complexity of FSA manipulation algorithms is usually exponential in the number of states, while other sources of inefficiencies (e.g., SPAF is coded in Java) are less important.

	pFSA			SPAF		
chosen filter	memory (MiB)	states	ratio (MiB/ state)	memory (MiB)	states	ratio (MiB/ state)
filter 1	37.248	4	9.312	1092.608	16	68.288
filter 2	37.472	4	9.368	1537.984	32	48.062
filter 3	37.776	5	7.555	1563.216	26	60.124
filter 4	39.088	9	4.343	1605.696	40	40.142
filter 5	38.384	4	9.596	1591.888	32	49.746

Table 2.2. Memory usage and number of states

Table 2.2 displays, for each filter, the maximum amount of memory required for the compilation process by pFSA and SPAF (which depends on the intermediate transformation of the automaton), and the number of states included in the *final* automaton. These results apply to the *core* database and, in case of pFSA, they include also the count of intermediate protoFSA states. These numbers prove that there is a clear difference between

the two implementations: even if the memory usage represents a *peak* measurement, while the number of states is measured at the *end* of the filter compilation, those numbers give a rough indication of the different efficiency of those algorithms.

2.7.2 Filter runtime performance

The next test aims at evaluating pFSA runtime performance. A single packet trace was created by extracting HTTP sessions from multiple real-world traces, taken in our University campus, for a final size of about 1 GiB. All filters in Table 2.1 were reused, adapting them to the syntax used by the specific packet filter, if necessary; filters 4 and 5 were slightly edited in order to let them match the most active sessions in the trace. We measured the number of CPU cycles needed to execute each filter with different packet filters; tests for pFSA and SPAF were run with the *core* and the *full* protocol database.

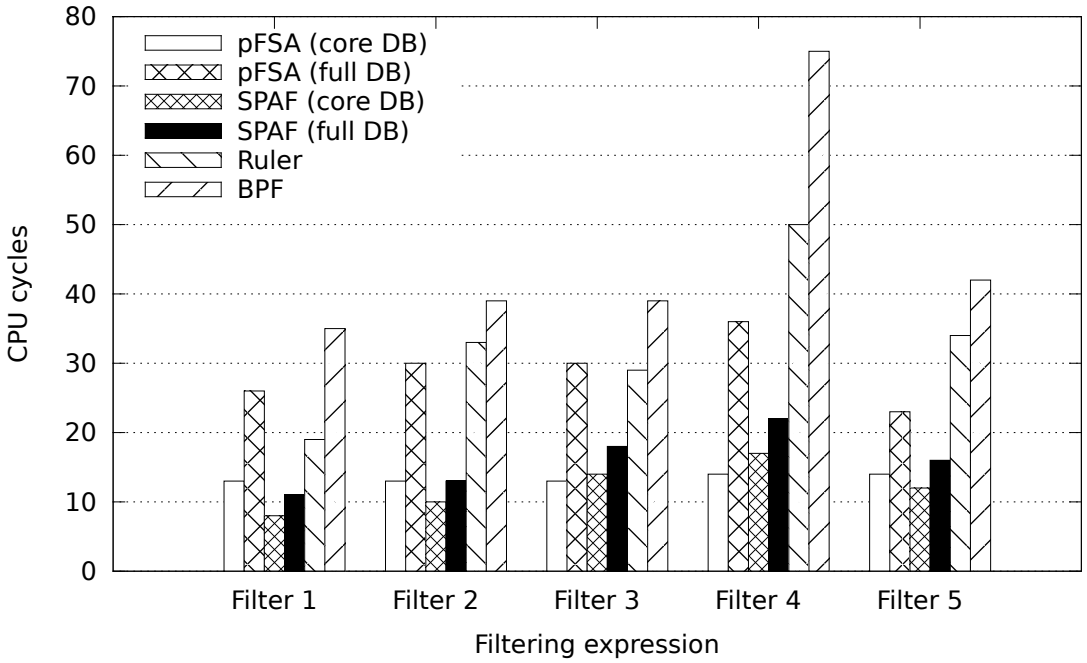


Figure 2.18. Maximum number of CPU cycles needed to evaluate a packet for each filter.

Figure 2.18 shows the *maximum* number of CPU cycles needed to analyze a packet, for each filter, implementation and (if applicable) protocol database. We have chosen to record the maximum number of cycles (instead of the average) to reduce the impact of non-matching and very short packets on the experiment. The best results are achieved by pFSA and SPAF: their performance is roughly the same, especially when using the *core* protocol database. SPAF leverages a more aggressive bounds checking algorithm that represents an advantage when many packet accesses are needed, such as in case of the *full*

database. In any case, pFSA results always faster than Ruler and BPF, even with the *full* database.

2.7.3 Filter scalability

The last round of experiments checks how pFSA performs when increasing the number of TCP sessions[11] in a given filter. Compilation times for filters with increasing number of sessions are tested first: results are shown in Figure 2.19.

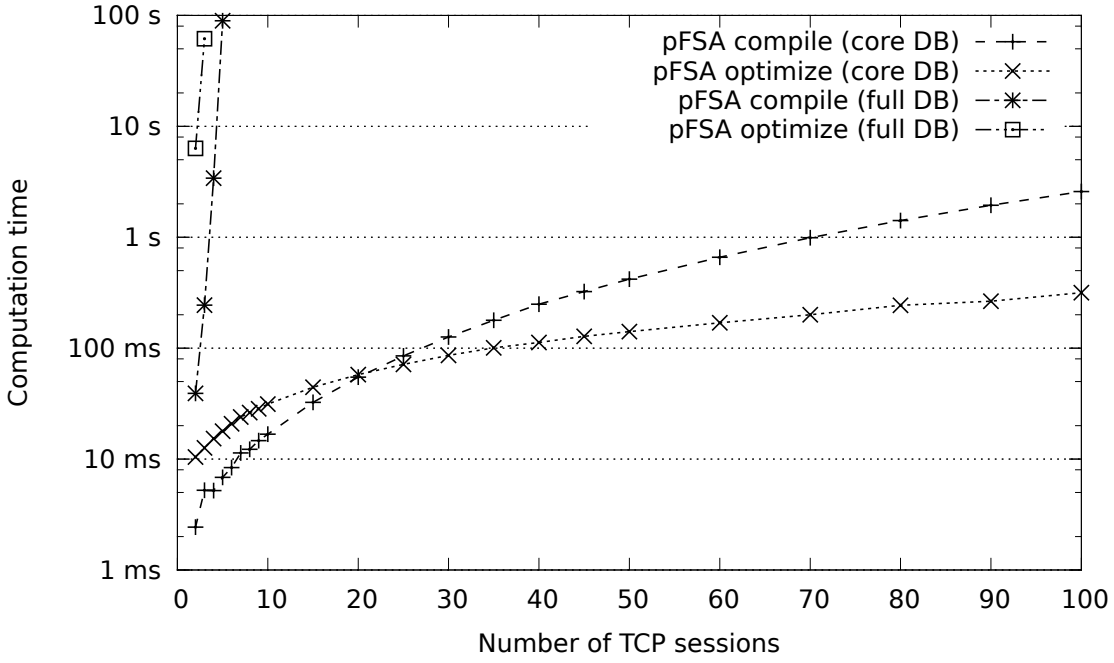


Figure 2.19. Compile and optimization times needed by pFSA to compile TCP session filters.

Our pFSA implementation was tested both with the *core* and the *full* protocol database. In the former case, the graph shows a more than linear, but still less than exponential increase in both compilation and optimization times. When the number of sessions is relatively low, the time spent optimizing the generated code prevails over the FSA generation time: but, since the generation time keeps growing faster than the optimization one, when the number of sessions increases over 20 the former overcomes the latter.

When the *full* protocol database is used, both compilation and optimization times grow exponentially in the number of sessions: for practical reasons, only the first data points are drawn in Figure 2.19. While unfortunate, this behavior is fully expected: the explanation

[11] In our example, a TCP session is defined as a uni-directional tuple of IP addresses (source and destination) and TCP ports (source and destination): e.g., filter 4 of Table 2.1 describes a single session.

lies in the filter statement and in the protocol database. When dealing with recursive encapsulations (e.g., `IPv4-in-IPv4`), a session filter, by itself, does not state that the IP source and destination addresses should both match inside the same IP protocol instance; e.g., filter 4 in Table 2.1 matches also a tunneled IP packet in which the outer IP source address is `10.1.1.1` and the inner IP destination address is `10.2.2.2`. Since a FSA does not have memory, the only way to handle this situation is by using different states for all possible combinations. When the number of sessions increases, the number of combinations (hence the number of states) grows exponentially, impacting computation times. However, it is worth mentioning that, should this behavior be undesired, the language we use to define the filter (NetPFL) includes additional primitives that, in presence of tunneling, allow to filter traffic based on a specific header of the packet (e.g., `ip%1.src == 1.1.1.1` to specify the source address of the first instance of IP *only*).

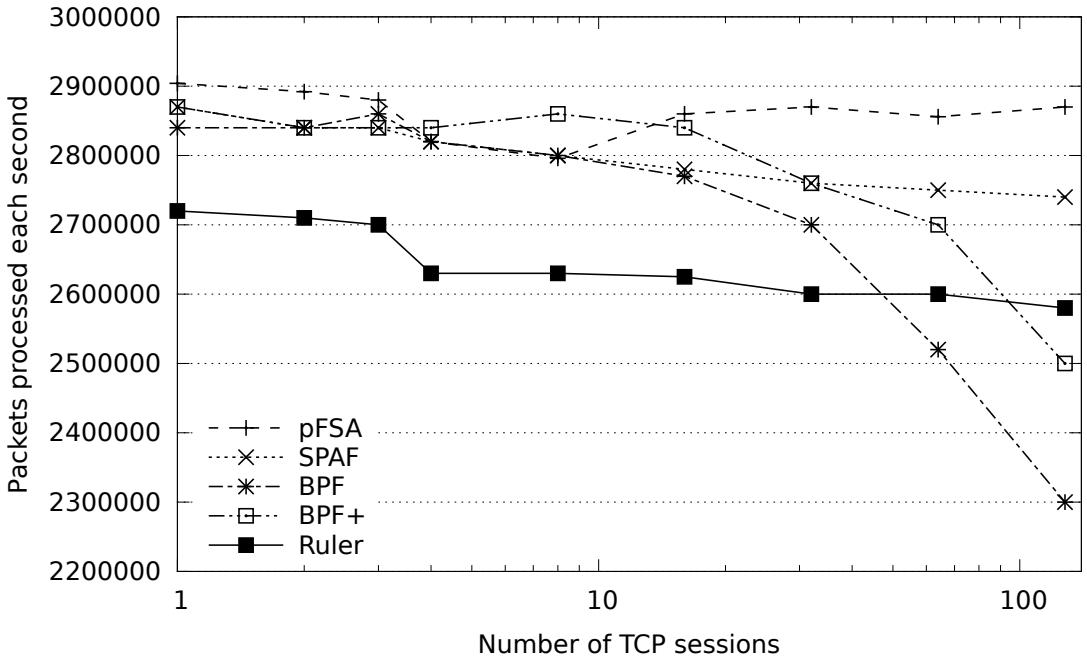


Figure 2.20. Overall runtime performance w.r.t. TCP session filters.

Our last test still focuses on TCP sessions scalability, but instead evaluates the runtime performance of pFSA. We used the same packet trace of Section 2.7.2 and tested the raw packet throughput: one-time computations (e.g., filter compilation) were not considered, but run-time overheads (e.g., per-packet libpcap library calls) are included in the results. Since some of the competing approaches cannot handle multiple levels of encapsulation, in this test we configured pFSA and SPAF to use the *core* protocol database. Figure 2.20 shows that pFSA does not suffer any significant runtime performance degradation when the number of filtered sessions increases. This is an expected scenario, because the generated

FSA grows wider, but not deeper; as the number of sessions grows, more and more states are added in parallel to the old ones, but the average distance from the starting state to the accepting ones does not change.

It is interesting to note the slight increase in performance just after the 16 sessions mark: the reason of this increase relies on the strategy that the NetVM JIT implementation uses to generate code for `switch` statements, emitted by the pFSA code generator to check for IP addressed and TCP ports. When the `switch` is sparsely populated and the number of cases is low (below 15), a Minimum Rectilinear Steiner Tree (MRST) is used; when the number of cases increases, a binary switch is used instead.

2.7.4 Ease of use

To conclude this section, we want to underline why pFSA is easier to use than many previous approaches, like BPF, specifically in case of complex protocol encapsulations.

Our pFSA-based implementation is able to generate filtering code that, according to the protocol database given in input, can match a filter for all possible encapsulations that can be recognized in the packet. If the protocol database contains the definition of a tunneled protocol, the pFSA model transparently filters it, without further input from the user.

As an example, let's imagine a scenario in which a given protocol database supports `IPv4-in-IPv4` tunnels. With a filtering string composed by only three tokens (i.e., `ip.src == 1.1.1.1`), pFSA and SPAF are able to match packets that contain at least one IPv4 instance whose source address is `1.1.1.1`, even if that instance is deeply nested in other protocols. A BPF filter like `ip src 1.1.1.1`, instead, is not tunnel-aware. To match the *first tunneled* IPv4 instance, a BPF user should write a filter that manually inspects the *protocol* field of the outer IP instance and then the IP address of the inner one, at the right offset: `ip[9:1] = 0x04 && ip[32:4] = 0x01010101`. Furthermore, for BPF to match both a "native" and the *first tunneled* IPv4 instance, both previous filters should be put in OR together: the number of tokens in the filter grows then to 11.

Number of encapsulations	pFSA	SPAF	BPF	BPF (when filtering at any level)
No levels	3	3	3	3
1 level	3	3	7	11
2 levels	3	3	11	23
3 levels	3	3	15	39
4 levels	3	3	19	59
5 levels	3	3	23	83

Table 2.3. Number of tokens in the filtering string needed to filter a tunneled IPv4 instance with a given destination address

Table 2.3 has a rundown of the increasing complexity (in terms of number of tokens in the filtering string) of a BPF filter, compared to the constant complexity required by pFSA and SPAF.

2.8 Conclusions

This chapter presented pFSA, a novel packet filtering model based on (multilevel) Finite State Automata augmented with predicates, which guarantees optimality of packet filtering composition with respect to the number of checks on the packet, even in case of complex predicates or unconventional protocol encapsulations, and independently from the complexity of the filtering string. Furthermore, being agnostic with respect to network protocols, our implementation exploits a dynamic protocol database that allows to change the protocols it operates upon by simply updating those files at run-time, without having to modify the source code of the packet filter compiler itself. Our model proved to be as fast as the best competitors for simple packet filters and to scale linearly with the number of predicates on the same protocol, such as when filtering multiple TCP sessions. At the same time it demands limited processing and memory requirements in the filtering code generation phase, which represents a huge improvement when compared with other approaches (e.g., SPAF).

Future work includes the capability to dynamically add and remove filtering expressions to an existing pFSA, which would allow to transparently optimize filters originated by independent applications without having to create multiple packet filters running in parallel, and a better integration of the pFSA model with the other components of the system (e.g., the protocol scanner). This will allow to keep the optimality property also when the model is translated into running code, while currently this is lost in our implementation during the lowering phase. However, in our experience the number of packet accesses is the minimum in most of the generated filters, although it cannot be guaranteed formally.

Chapter 3

xpFSA: efficient support for tunneled protocols

3.1 Introduction

In the recent years we have observed a reduction in the number of layer-7 protocols in use. While in the past each application defined its own protocol, nowadays most of the traffic is conveyed through the web, and HTTP has become the de-facto protocol for many different applications. Surprisingly, the opposite phenomenon was observed at the bottom of the protocol stack. While protocol encapsulations were definitely simple in the past (IP in Ethernet was by far the most common encapsulation), new necessities, arising in particular from network virtualization, are rapidly transforming the lower layers of the protocol stack. Figure 3.1 presents one of the possible examples of the complexity growing over the years, which, e.g., translates into frames that need several more fields to transport a simple IP packet, compared to what it was defined in the original Ethernet DIX specification in the early '80s.

When using tunneling or network virtualization protocols, packet filtering becomes more complicated: it is important to be able to capture all the traffic we are interested in (e.g., web traffic), independently from the actual encapsulations used at lower layers. At the best of our knowledge, no packet filter implementation available today offers a flexible tunneling support: to support more complex protocol encapsulations, one has to modify the source code of the filtering tool itself.

NetPFL, described in Section 1.3, aims at solving this problem, by providing an easy-to-use filtering language that supports protocol tunneling with a flexible syntax. Most of its features have already been implemented in `nbeedump`, a dynamic packet filter generator part of the NetBee library [30]; NetPFL *tunneling* support, however, is missing from `nbeedump`, because the pFSA model, described already in Section 2, is not generic enough to model tunnels with good performance in all circumstances.

In this chapter we propose a model for packet filter called **xpFSA** (or *Extended Finite State Automaton with Predicates*), that improves the previous model by extending its support for tunneling features.

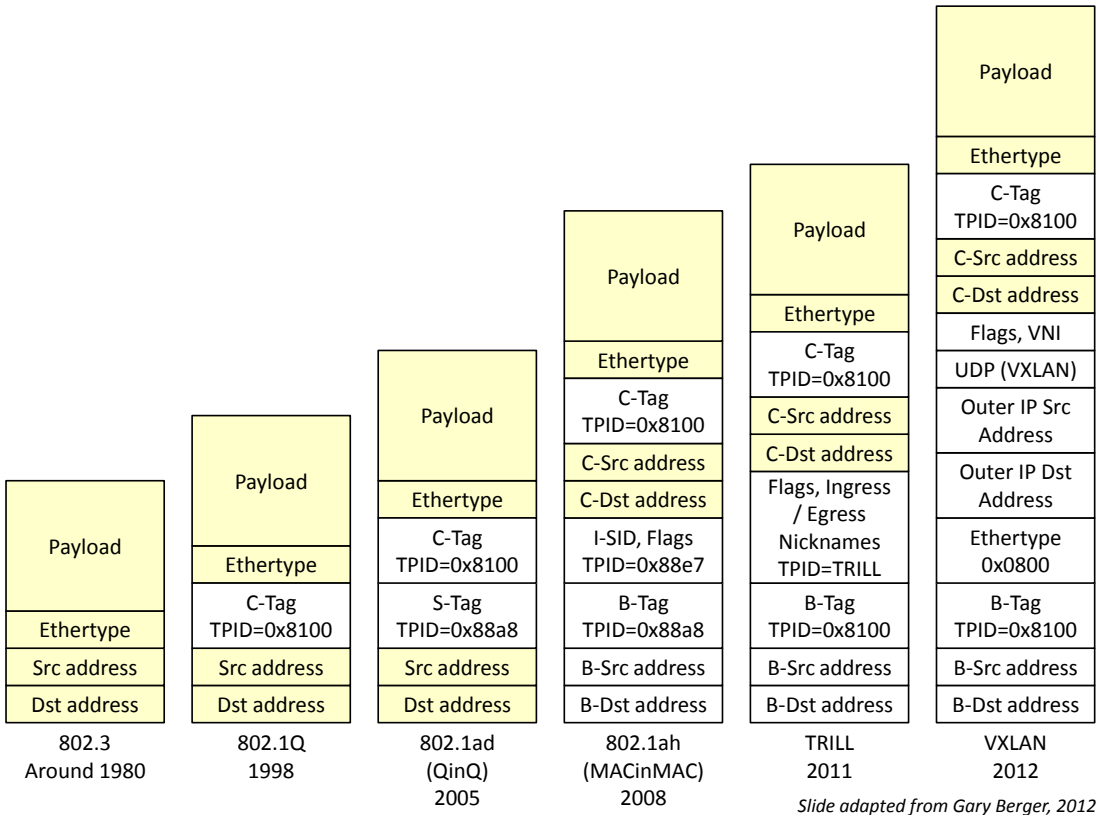


Figure 3.1. Growing complexity in protocol encapsulations.

Section 3.2 presents the state of the art; Section 3.3 presents the xpFSA model, along with the algorithmic details; Section 3.4 shows how to transform a NetPFL statement into an xpFSA automaton. Section 3.5 presents the algorithms required to support NetPFL's `tunneled` keyword, while Section 3.6 explains how xpFSA can also be used for packet field extraction. Section 3.7 describes a minor extension to the NetPDL language to make the default behavior more user-friendly. Finally, Section 3.8 presents an overview of our implementation, leaving the experimental evaluation to Section 3.9 and conclusions to Section 3.10.

3.2 Related Work

The work of this chapter extends the pFSA model in order to represent (complex) filtering expressions based on protocol encapsulation rules. Despite the high number of publications on packet filters, at the best of our knowledge none of them proposes a solution able to handle filtering conditions with encapsulation constraints. For example, neither libpcap [8], representing the foundation of many packet filtering tools (e.g., `tcpdump`,

Wireshark), nor the display filters implemented in Wireshark [3] (which replace the basic filtering capabilities of libpcap when packets have to be shown on screen) support filters with tunneling features.

The idea of using code to reduce the number of states has been mainly influenced by the **Extended Finite Automata** (XFA) [27] formalism, that augments traditional FSA with finite scratch of memory and generic executable code to manipulate this memory. XFA associates code with states and transitions, but, while, its primary goal is to improve the time and space efficiency of signature matching in network intrusion detection systems, our work aims at supporting tunneling features in packet filters applications.

Similarly, the **Extended Finite State Automata** (EFSA) [26] formalism extends traditional FSA to assign and examine values of a finite set of variables, in order to model fast intrusion detection and prevention systems. Even if EFSA has a different goal than our model, both formalisms augment transitions through predicates that evaluate the value stored in a variable. In spite of this, EFSA also associates with transitions the code to store values into variables and supports actions associated with final accepting states.

Stateless FSA-based Packet Filter (SPAF) [13] is a packet filter generator based on the creation of finite state automata from high level protocol database and filter predicates. Similarly to our work, it enables full parsing of complex protocols and supports recursive encapsulation relationships. However, each protocol is first modeled through a FSA that consumes a byte at a time and then, by using classical rules from literature, the various automata are combined together in order to represent the whole filter. Although SPAF guarantees code optimality and safety, it is extremely slow in the packet generation phase, because the number of generated states is very high, and consequently the determinization phase takes a lot of time.

Except SPAF and pFSA, we are not aware of other packet filtering models based on a FSA-like approach. Even if both of them support multiple instances of the same protocol, and both take into account a Protocol Encapsulation Graph (PEG) in order to represent a filter, we chose to extend the pFSA formalism because it generates much more compact (and efficient) automata.

For some kinds of filters, however, the pFSA model could lead to automata with a very high number of duplicate states. An example is shown in Figure 3.2, where we represent a pFSA modeling a filter that requires the IP protocol to appear *at least three times* within a packet (i.e., NetPFL filter `ip%3`). Our aim is to enhance the pFSA formalism, in order to reduce the number of duplicate states, and consequently to have better compile-time performance.

3.3 Extended FSA with Predicates

This section presents the **xpFSA**, an extension of the pFSA model that improves its predecessor by extending its support for tunneling features.

Briefly speaking, the input symbols can be associated with operations on specific counters, and the transitions can have predicates expressed on the value of these counters. This way, the number of duplicate states is reduced, and consequently the complexity of the

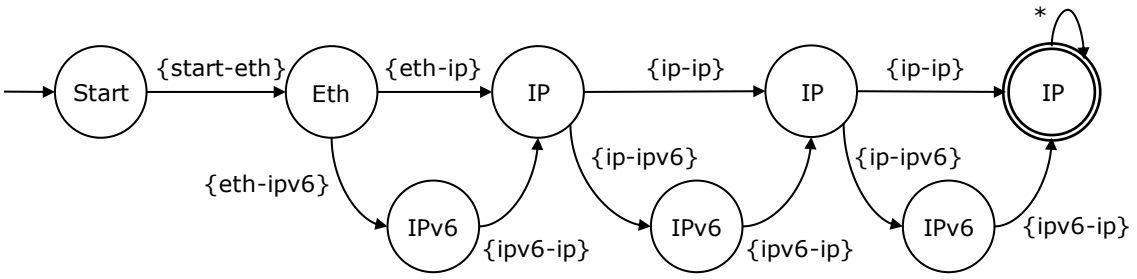


Figure 3.2. Example of a pFSA with duplicate states.

generation and composition algorithms. In addition, in order to model filters requiring generic actions, xpFSA supports *action states*. More details are given in the following sections.

3.3.1 Definition of xpFSA

An **Extended Finite State Automaton with Predicates** (**xpFSA**) is defined with the following "eight-tuple":

$$A_{xpfsa} = (Q,\ \Gamma,\ A,\ \Sigma_c,\ \delta_p,\ q_0,\ F,\ E)$$

where:

Q is a finite set of *states*, each of which is associated with a specific network protocol[1];

Γ is a finite set of *integer counters*;

A is a finite set of *generic actions*;

Σ_c is the set of *input symbols*, each of which is associated with a (potentially empty) list of *counters*. Each symbol represents a protocol encapsulation rule, while the list contains information about the increase and/or decrease of one or more counters among those in Γ. When the control of the automaton receives a symbol, before triggering the proper transition(s), performs also the required operations(s). Obviously, if the list associated with the received symbol is empty, the control of the automaton behaves as in a pFSA;

δ_p is the *transition function with predicates* (p-transition). It is similar to the one defined for pFSA, but predicates can also be expressed on the value of the counters

[1]There are two exceptions to this rule: *(i)* the starting state is associated with a "dummy" protocol in the database, since it is used to represent the state of the automaton before the analysis has begun; *(ii)* most pFSA have a state representing the non-accepting condition, reached if the processed packet does not satisfy the filter: that state is not associated with any network protocol.

defined in Γ. Note that it is possible that the predicate is always true, and in this case the p-transition is equivalent to a classical transition defined in the base FSA model;

q_0 is the *starting state*, among those in Q;

F is a set of *accepting states*, among those in Q;

E is a set of *action states*, among those in Q. When one of them is reached, specific actions among those in A are executed.

Figure 3.3 shows the xpFSA representing the NetPFL filter `ip%3`, which requires that a packet, in order to be valid, contains at least three instances of IP.

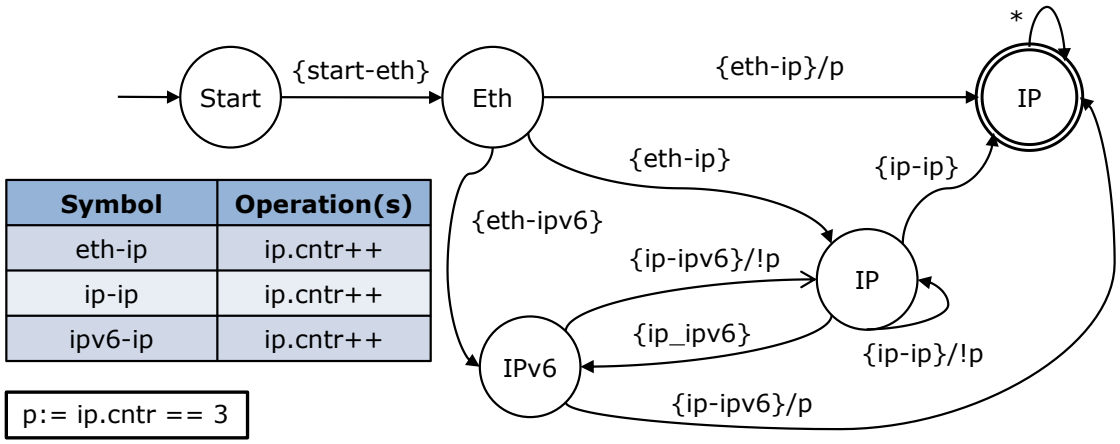


Figure 3.3. Example of xpFSA.

In the example, the operation `ip.cntr++` is associated with all symbols having IP as target protocol and then, each time that one of them is received by the control of the automaton, the variable `ip.cntr` is incremented by one. Moreover, the example xpFSA has some p-transitions which evaluate the value of this counter, in order to understand if the final accepting state can be reached and consequently if the packet can be accepted.

An important consideration is that, even if both Figures 3.2 and 3.3 represent the same filtering expression, the second one has a single state related to IPv6, instead of two, and two states associated with IP, instead of three: xpFSA reduced the number of duplicate states. Furthermore, by tweaking only the value against which the automaton in Figure 3.3 evaluates the variable `ip.cntr` in predicate `p`, all filters in the form `ip%n` can be represented.

It is worth pointing out that a counter associated with a symbol is incremented (or decremented) as soon as the symbol is received, but before the predicates are evaluated. As a consequence, if a transition with predicates evaluates the value of a counter incremented by the symbol related to the transition itself, the evaluation is influenced by the operation on the counter.

3.3.2 Determinism

The determinism of an automaton is important for different reasons. For example, it is required for FSA complementation, which in turn is required by automata intersection, if the latter is implemented using first De Morgan's law ($A \cap B = \overline{\overline{A} \cup \overline{B}}$). Moreover, a deterministic automaton can be translated into executable code more easily than a non-deterministic one: the former in fact can have only an active state at a time, while in the latter either the control could be in multiple states in the same instant, or runtime backtracking must be implemented.

The definition of **deterministic** pFSA depicted in Section 2.3.3 is still valid in the new model. In particular, an xpFSA is deterministic if it does not include any ϵ *transition*[2] and, for each input symbol and for all possible values of the Boolean predicates, there is exactly one enabled, outgoing transition.

3.3.3 Algorithms

The algorithms defined in the pFSA formalism (Section 2.3.4) require some changes in order to be reused in our new model. This is due to both the operations associated with the symbols, and the newly added feature of *action states*.

The **union** algorithm can be reused with little effort. In fact, since it merges two automata by adding a new initial state, connected to the initial states of the original automata using ϵ transitions, action states are not considered. On the other hand, the counters and the related operations associated with the input symbols are involved. In particular, the set of variables of the resulting automaton is the union of the sets associated with the two joined automata. Each of the input symbols of the resulting automaton is then associated with the union of the lists of operations assigned to the same input symbol in both original automata, excluding duplicates. This means that, for example, if in both automata to be joined a counter is incremented on the same symbol, that symbol will be associated with a single operation in the resulting FSA; the same rule applies when the counter is decremented. However, since increment and decrement operations can coexist, they can effectively cancel each other out.

Counters are not considered in the **complementation** algorithm, since it requires only to reverse the accepting status of all states of a deterministic automaton. Consequently, an action state, regardless of whether it is accepting or not, is still an action state after the complementation, while a non-action state continues to be a non-action state.

Thanks to first De Morgan's law, the **intersection** algorithm does not require extra effort, because it is based on the complementation and union algorithms.

It is worth noting that both the union and the intersection algorithms may produce non-deterministic automata with redundant states. For this reason, further operations are needed in order to create better automata: **determinization** and **minimization**.

The determinization algorithm is a little different with respect to the one depicted

[2]Please remember that an ϵ transition does not require any input symbol to fire.

in Section 2.3.4, because of the introduction of action states. In detail, if N is a non-deterministic xpFSA, and D is its equivalent deterministic automaton, the set of action states of D consists of all the states that, during the determinization algorithm, were created out of at least one of the action states of N. Moreover, the set of actions to be executed when an action state of D is reached is the union of all actions associated with the states of N from which the resulting state comes from. An example, that for the sake of simplicity does not have any reference to network protocols, is shown in Figure 3.4. In particular, the non-deterministic automaton is in the left of the figure, while the deterministic one is depicted to the right. The state $\{q_0, q_1\}$, as highlighted with the dashed circle, is an action state, because q_1 was an action state in the non-deterministic xpFSA.

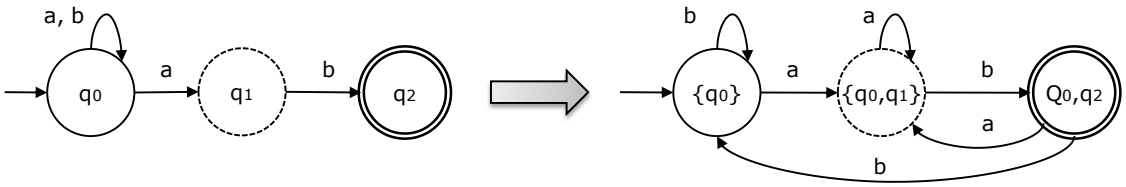


Figure 3.4. Example of determinization of an xpFSA.

Consider now the minimization algorithm. It requires to identify equivalent states and compact them into a single state. From the automata theory [21], two states could be equivalent if they have the same status, i.e. if they are both accepting or both non-accepting. This is still true in an xpFSA, but now there are four possible statuses: *(i)* accepting and action; *(ii)* accepting and non-action; *(iii)* non-accepting and action; *(iv)* non-accepting and non-action. When the minimization algorithm merges two states, the resulting one executes both actions associated with the original states. Moreover, as defined in Section 2.3.4, two states are equivalent or not depending on their outgoing transitions. An example of minimization of an xpFSA is shown in Figure 3.5. Note that q_4, represented with a dashed double circle, is an accepting and action state.

To conclude, it is worth noting that: *(i)* neither the determinization nor the minimization consider the counters and the operation associated with the symbols; *(ii)* no algorithm takes into account the state-protocol association, needed to translate the xpFSA into executable code to analyze network packets.

3.4 Building the xpFSA

This section presents the algorithm used to build the xpFSA representing NetPFL header chains, starting from the protocol encapsulation rules represented in a PEG. This algorithm extends the one defined in the original pFSA formalism (Section 2.4.5) because that algorithm: *(i)* was not able to create an automaton representing an header chain, but

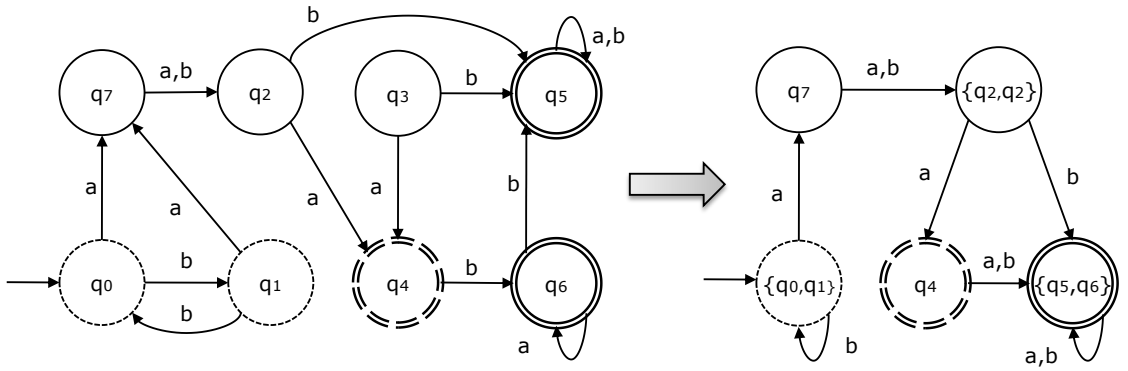


Figure 3.5. Example of minimization of an xpFSA.

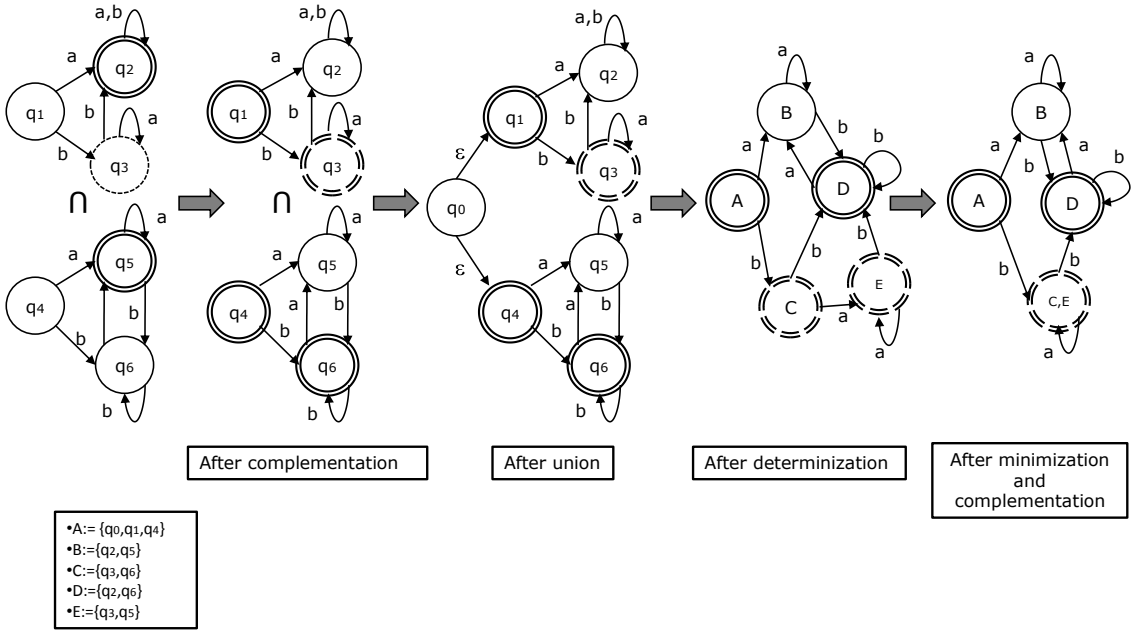


Figure 3.6. Example of union of two xpFSA.

only pFSA related to simple filters like `tcp.sport==80`, possibly joined with Boolean operators; *(ii)* did not support both operations associated with the symbols, and predicates evaluating the value of counters, required to reduce the number of states of the automaton.

In practice, the concepts defined in the old algorithm to create a pFSA are still used: *(i)* to join the automata representing different header chains, and *(ii)* to create the sub-automata associated with the pairs of p-transitions.

3.4.1 NetPFL to regular expression

A preliminary step of the algorithm consists in the translation of the header chain into a regular expression, whose alphabet is composed by the protocols associated with the states of the referred PEG. It is worth noting that this regular expression encodes only some aspects of the filter, because the predicates on fields, the header indexing and the keyword `tunneled` do not have a representation in the regular expression formalism. These aspects of the header chain are not considered in this preliminary stage of the algorithm, but they will be taken into account in later steps.

In order to create the equivalent regular expression, each NetPFL `token`, starting from the right-most one, is translated by applying the rules depicted in Table 3.1.

NetPFL	RegExp
`in {p1,p2}`	`[p1 p2]`
`notin {p1,p2}`	`[^p1 p2]`
`in any`	`.`

Table 3.1. Translation rules

An important consideration is that this table does not show how to convert the repeat operators (i.e. $+$, $*$ and ?), because they are represented in the same way in both formalisms. After the translation of the `token`s, the `target` element of the header chain is placed in the rightmost position of the regular expression. The regular expression is then extended at both ends with the element $.*$, which matches any protocol repeated an unbounded number of times. This way, that sequence of protocols can appear everywhere within the packet[3].

As an example, consider the NetPFL header chain:

```
tcp.sport==80 in ip%2 in {ip,ipv6}+
```

which becomes the regular expression:

```
.* [ip ipv6]+ ip tcp .*
```

It is worth nothing that, in line with the rules introduced above, neither the predicate on the destination port of TCP, nor the header indexing requirement for the IP header, are represented into the regular expression.

This last remark highlights that there is not a one-to-one correspondence between the header chain and regular expressions, since some NetPFL elements are not considered in the transformation process: the transformation from a NetPFL filter to a regular expression is not a *injective* function, hence multiple filters can be represented with the same regular expression. However, as it will become clearer in the following, this does not represent a problem, because the NetPFL elements not represented in the regular expression will be later taken into account.

[3]We can optimize filters having `startproto` in the right-most position: since `startproto` is a dummy protocol representing the beginning of the packet, in the equivalent regular expression the first $.*$ is omitted.

3.4.2 The skeleton of the automaton

The goal of this stage of the algorithm is to create an automaton out of the regular expression just built, and then associate each of its states with one or more protocols.

In particular one element of the regular expression is considered at a time, from left to right, and, depending on its repeat operator, a different block of the automaton is created. The translation rules are depicted in Figure 3.7, and they are derived from the standard mapping rules defined from the automata theory [21]. We can optimize the last element of the regular expression, i.e. `'.*'`, for which a looping transition on the last state, firing on any symbol, is enough.

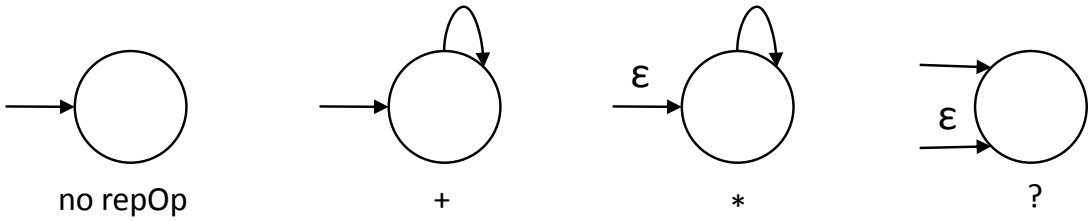


Figure 3.7. Building blocks of the automaton.

All automaton blocks are then connected in order, and the right-most state represents the final accepting state of the automaton.

Each state, if it does not have any incoming ϵ transitions, is associated with the protocol(s) specified by the element of the regular expression from which it derives. Otherwise it is also associated with the protocols related to the origin state of the ϵ transition. This is because, whenever the automaton control is in the origin state of the ϵ transition, it must be also in the target state of the transition itself; therefore its destination state is reached also when a protocol leading to its source one is encountered within the packet.

For example, consider the regular expression `.* ethernet .+ tcp .*`, coming from the NetPFL header chain `tcp.sport in any+ in ethernet`, and translated into the automaton shown in Figure 3.8. That figure, in addition to the automaton, shows also the element of the regular expression from which each building block derives (at the top), and the protocols represented by each state (in the grey boxes at the bottom).

At this point the skeleton of the automaton has been completed, according with the information represented in the regular expression. However, the automaton is still incomplete, because: *(i)* the transitions are not labeled with any input symbol yet; *(ii)* any state could still be associated with multiple network protocols; *(iii)* the NetPFL elements which have not been coded into the regular expression must still be considered.

3.4.3 Defining the counters

This step of the algorithm aims at defining the set of counters of the xpFSA, and at associating operations on these variables with the input symbols. To reach this goal, the

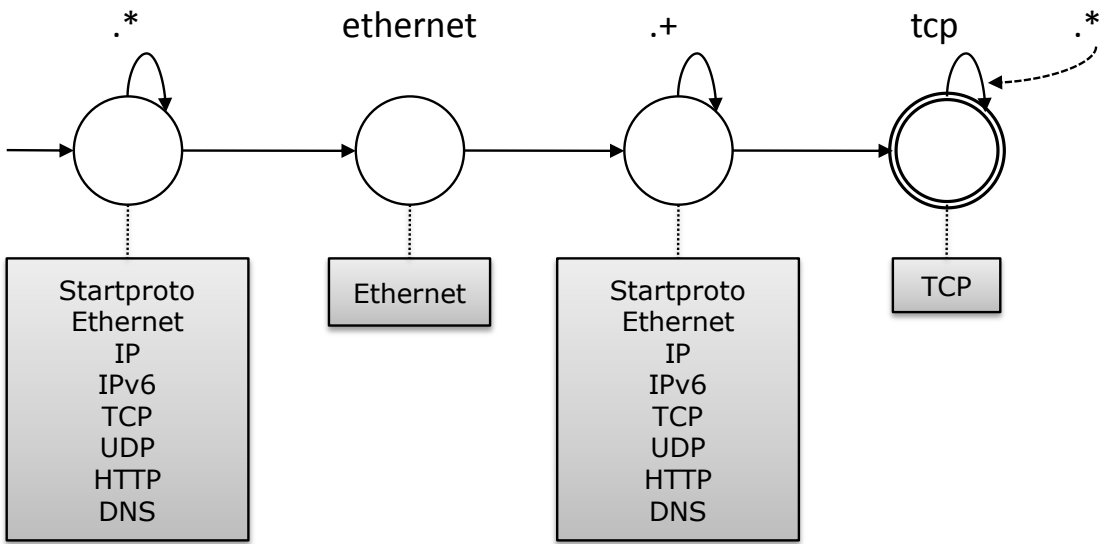


Figure 3.8. Skeleton of an automaton representing a header chain.

optional header indexing features, defined within the header chain, are considered[4].

In particular, for each header indexing related to a different protocol, a new variable is created, with a name in the form `protoID.cntr`. Moreover, in order to associate the operations related to these variables to the appropriate symbols, the following rule is applied: *an operation of increment of a counter related to a certain protocol is assigned to all symbols having that protocol as target.*

Considering for instance the filter `ip%2`, the variable `ip.cntr` is created, as it is needed to count the number of IP headers encountered within the packet under analysis. Then, the operation `ip.cntr++`, which increments `ip.cntr` by one, is associated with those input symbols having IP as target, i.e. `eth-ip`, `ip-ip` and `ipv6-ip`.

3.4.4 Labeling the transitions

At this point, the algorithm associates each *non-ϵ* transition of the skeleton of the automaton previously created with one or more input symbols, in order to transform the skeleton itself into an actual FSA (but not yet into an xpFSA).

In particular, a transition is labeled with all the symbols having the name satisfying the following constraints: *(i)* the first part, i.e. the origin protocol, is equal to one of the protocols associated with the source state of the transition itself; *(ii)* the second part, i.e. the target protocol, is equal to one of the protocols specified by the element of the regular expression from which the destination state comes from. This means that protocols

[4]The NetPFL keyword `tunneled` needs specific considerations, so it will be discussed later, in Section 3.5.

associated with a state because of the ϵ transition cannot be target of the symbols leading to the state itself, since this could cause the recognition of wrong packets.

To better understand this statement, consider the fragment of regular expression `ipv6 ip*`, resulting in the piece of automaton shown in the left of Figure 3.9. The rightmost state is associated with IP and IPv6 but, as emphasized with the square brackets, since the IPv6 association is due to the ϵ transition, IPv6 cannot be the target protocol of the symbols on the self loop. This way, this automaton excerpt recognizes only the sequences of protocols like IPv6, IPv6 - IP, IPv6 - IP - IP and so on. Instead, if IPv6 was the target of the symbols on the self loop, the automaton would also accept sequences such as IPv6 - IP - IPv6 - IP, that do not satisfy the fragment of regular expression `ipv6 ip*`.

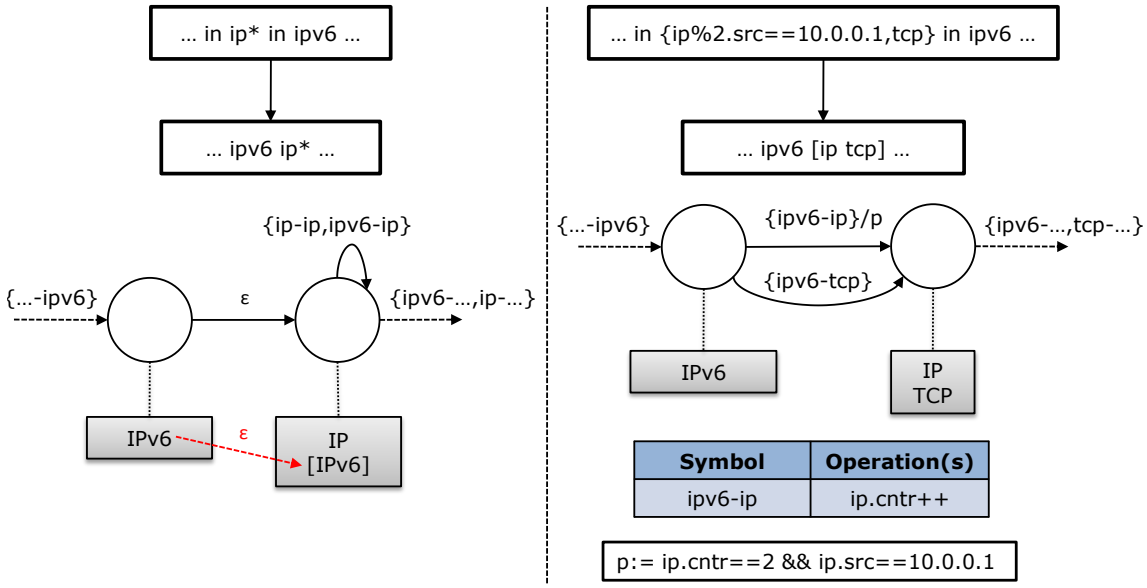


Figure 3.9. Transitions labeling process.

Obviously, since we are building an xpFSA representing a NetPFL header chain, it is possible that some transitions evaluate some predicates. As an example, consider the right part of Figure 3.9, representing the piece of header chain `'in {ip%2.src==10.0.0.1,tcp} in ipv6'`. Because of the predicates on the source address and on the number of encountered IP headers, a p-transition toward the rightmost state is needed. It is important to note that the transition fires if the condition `ip.cntr==2 && ip.src==10.0.0.1` is verified, because this predicate comes from a NetPFL element involved in the `in` operation. On the contrary, if this element had been used in a `notin` operation, the p-transition would have fired if `(ip.cntr==2 && ip.src==10.0.0.1)` returned `false`.

After the labeling process just described, the transitions of the automaton of Figure 3.8 are labeled as shown in Figure 3.10. Note that a p-transition links the third state with the final accepting state of the automaton, because of the predicate specified on the source port of the TCP header.

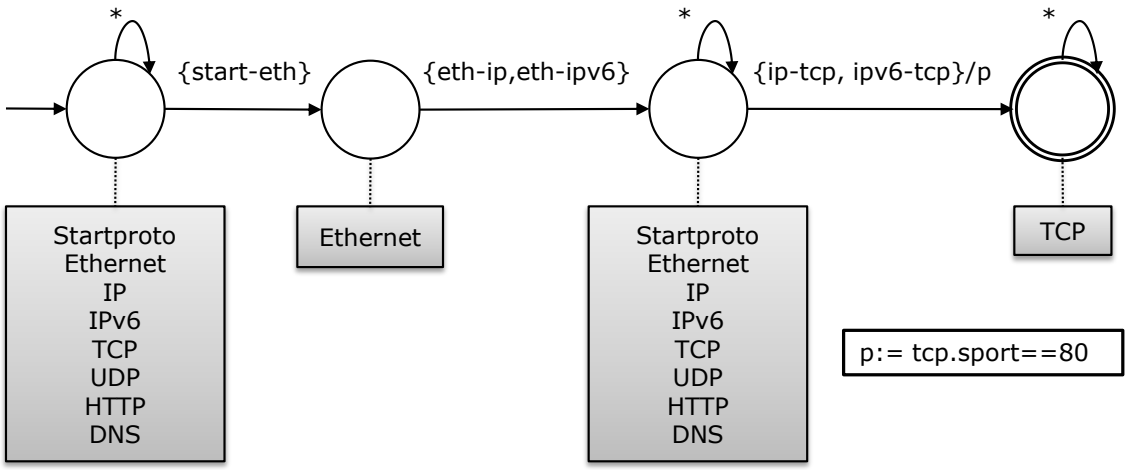


Figure 3.10. Automaton with labeled transitions.

3.4.5 The automaton representing the header chain

By observing Figure 3.10, it is evident that the automaton just built could be non-deterministic. For this reason, it is now translated into a deterministic one using the algorithm defined in the xpFSA model, becoming as shown in Figure 3.11.

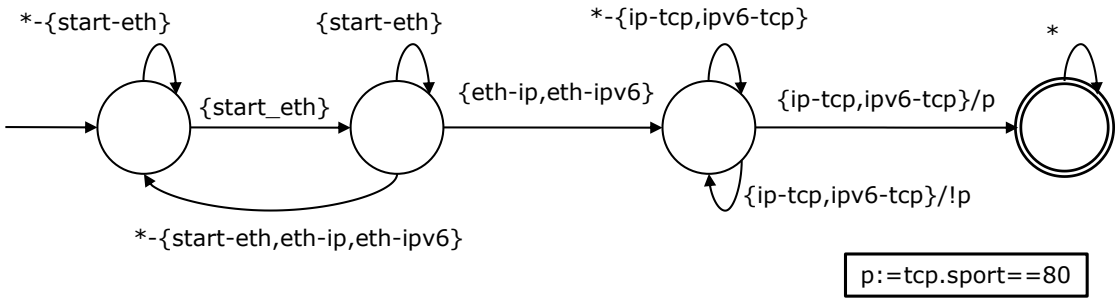


Figure 3.11. Deterministic automaton representing an header chain.

It is worth noting that, even if the automata in Figure 3.10 and 3.11 are already representing a NetPFL header chain, they cannot still be used for packet filtering, as this requires that each state is related to a single protocol; but, in the previous algorithm steps, each state may have been associated with multiple protocols. Moreover, the state-protocol association might have been lost during the determinization process[5]. As a consequence,

[5]In fact Figure 3.11, unlike the previous ones, does not show any correspondence between protocols and

the next steps of the algorithm manipulate the automaton until each state is associated with one and only one protocol, so that we can guarantee that reaching a certain state of the automaton corresponds to reaching a specific protocol within the packet under analysis. This is needed to later be able to translate the automaton into executable code that can analyze network packets.

3.4.6 Managing states already representing a single protocol

This step of the algorithm aims to identify states already representing the reaching of a single protocol, and to label each of them. In detail, a state corresponds to a specific protocol if all the symbols associated with its incoming transitions share the second part of their name, i.e. the target protocol of the encapsulation rules that they represent is the same. Two exceptions are: *(i)* the initial state, which represents a single protocol (i.e. Startproto) only if it does not have any incoming transitions, and *(ii)* the accepting state, whose self loop is not considered.

After that a state has been labeled, the symbols on its outgoing transitions are removed if their origin protocol differs from the one represented by the state itself[6]. Obviously, when a transition remains without symbols, it is removed. Also, a state is removed if it is disconnected from the other ones.

These operations are repeated until there are no more changes in the automaton. After the execution of this step, the automaton of Figure 3.11 becomes as shown in Figure 3.12.

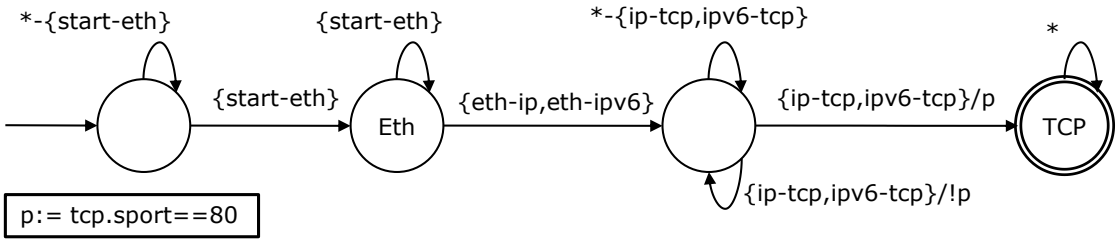


Figure 3.12. Automaton with some labeled states.

3.4.7 Expanding states and transitions

At this point, the algorithm continues by splitting each unlabeled state in multiple states, each one associated with a different protocol among those that are the target of the symbols assigned to the transitions entering into the former state. It is worth noting that, in order

states.

[6]This is possible because the symbols represent protocol encapsulation rules. Hence, if the current state is associated with IP, only the symbols representing a protocol encapsulated into IP can be received while the xpFSA is in that state.

to represent the situation in which the analysis of the packet has not started yet, we need to associate a state with Startproto. However, no input symbol exists with this protocol as a target, then the initial state is also expanded in one associated with Startproto and representing the new initial state of the automaton. As an example of expansion, consider the dark state in the left of Figure 3.13, which originates two new states, respectively representing the protocols IP and IPv6 (shown in the right of the figure).

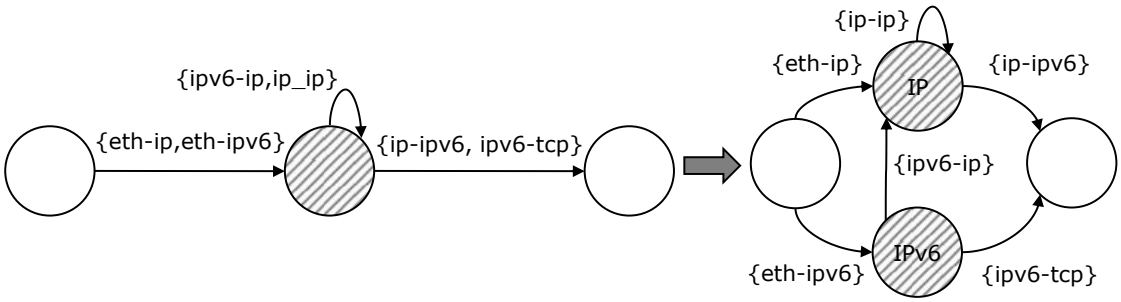


Figure 3.13. Expansion of a state and the related transitions.

Each transition *exiting* from an expanded state is replaced with a new transition for each one of its symbols. In particular, each of these new transitions starts in the new state representing the source protocol of its symbol, and terminates in the same state of the original transition. For example, the transition exiting from the dark state in the left of Figure 3.13 originates two new transitions: one labeled with `ip-ipv6` and coming from the new state associated with IP, the other firing with `ipv6-tcp` and originating in the new state representing IPv6.

Similarly, the transitions *entering* into an expanded state are replaced based on the target protocols of their symbols. This way, the transition labeled with `eth-ip` and `eth-ipv6` originates two transitions: one firing with `eth-ip` and leading to the new state representing IP; the other labeled with `eth-ipv6` and entering into the new state associated with IPv6.

Figure 3.13 shows also that the *self loop* on an expanded state originates a new transition for each one of its symbols. In particular, a new transition starts and ends on the proper new states, according with the origin and target protocol of the symbol associated with it.

Figure 3.14 shows how the automaton of Figure 3.12 becomes after this step of the algorithm. Note that the symbols on the new transitions have not been specified for reasons of brevity, because they can be easily derived from the protocols labeling the states.

3.4.8 The xpFSA representing the header chain

Figure 3.14 shows an xpFSA in which each state corresponds to a single protocol: hence, it can be translated into executable code to analyze network packets.

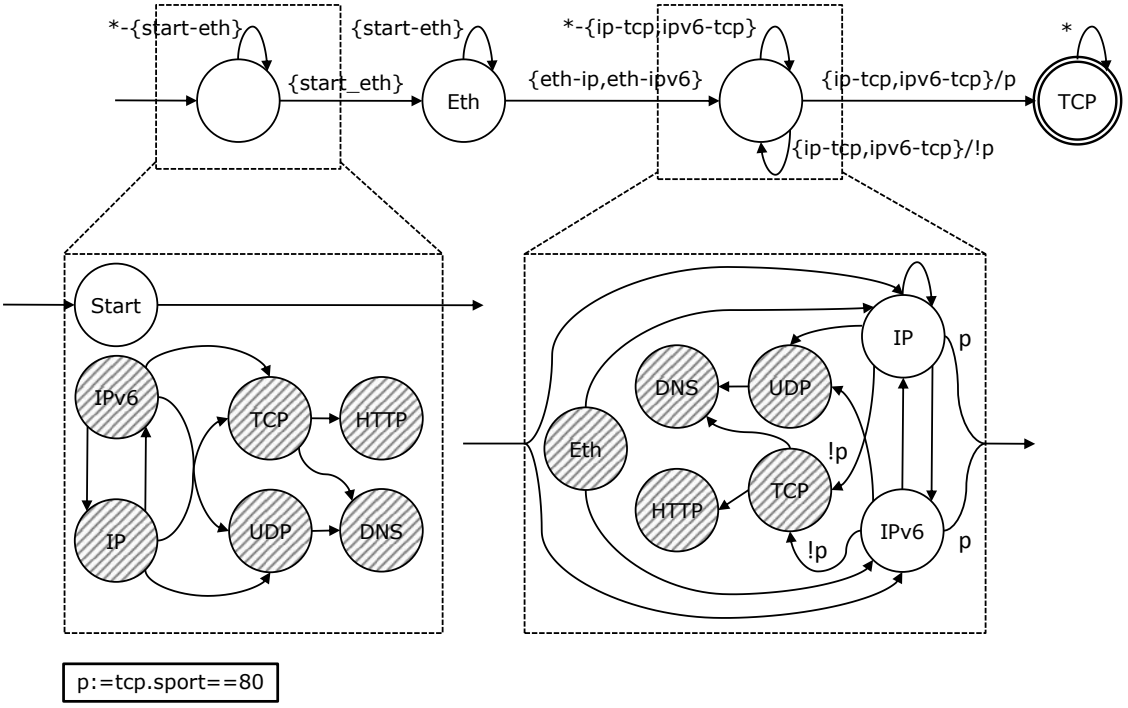


Figure 3.14. Expansion of unlabeled states.

It is worth noting that the xpFSA could have more states than the minimum necessary to represent an header chain. For instance, the dark states in Figure 3.14 are useless, either because they are never reached, or because they do not allow the control to reach the final accepting state: consequently, these states could be removed without any change in the meaning of the automaton. In order to identify these states, a reverse post-order visit of the xpFSA is performed, starting from the final accepting state; states that cannot be visited are removed.

After pruning useless states and transitions, the xpFSA of Figure 3.14 becomes the automaton shown in Figure 3.15, which is finally the minimized xpFSA.

To conclude, it is worth remembering that a NetPFL filter could be the composition, through the Boolean operators `and` and `or`, of multiple header chains. In this case, our algorithm is executed for each of them, and the resulting automata are joined using the algorithms defined in the xpFSA model.

3.5 Identifying tunneling

This section describes the algorithm to create the xpFSA recognizing packets that satisfy an header chain, if a `tunneled` protocol (or set of protocols) is required by the filter.

In this context, the following definition of "tunneling" is used: *a protocol header is*

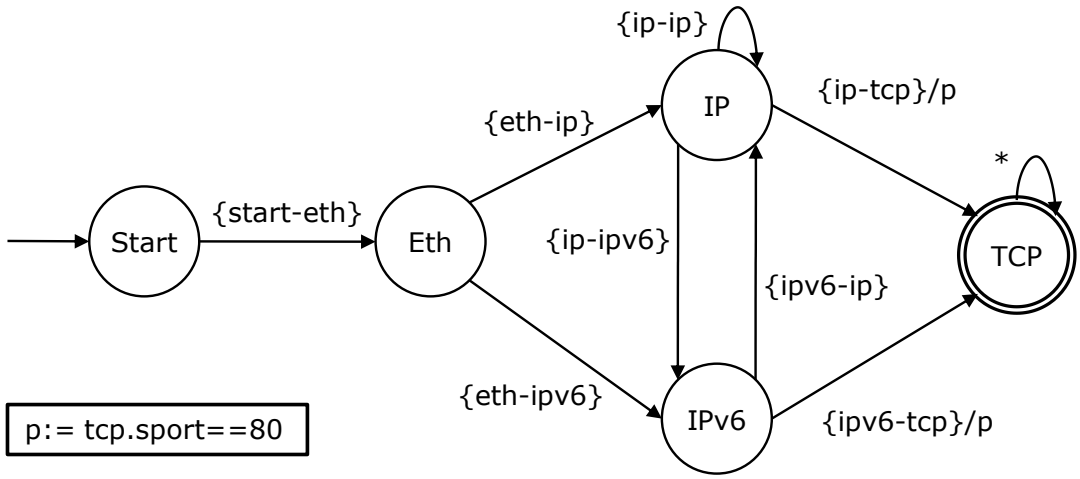


Figure 3.15. xpFSA representing a NetPFL header chain.

tunneled if the layer of at least one of the protocols headers preceding it in a packet is greater than, or equal to, the layer of the protocol that is being considered.

The xpFSA recognizing packets with tunneled protocols, as it will be detailed later, uses a variable that counts the number of protocols encountered within a packet and belonging to certain network layers. Consequently, to understand when to increment this counter, it needs to know the layer associated with each protocol defined in the PEG in use. For this reason we define a procedure that, given a PEG, assigns a network layer to each protocol.

3.5.1 Assigning layer numbers to protocols

For many protocols, their layer can be inferred from the traditional ISO/OSI protocol stack. For example, Ethernet belongs to layer 2, IP and IPv6 to layer 3, while TCP and UDP belong to layer 4. However, there are protocols that do not fit well in this model: for those protocols, choosing the "right" layer is debatable. A good example is MPLS, which may be present between Ethernet and IP. Therefore, it can be considered as belonging to layer 2, to layer 3 or to an intermediate layer.

Therefore, labeling each protocol with a number indicating its "natural" layer can be a complex operation. Furthermore, a previous labeling might not be valid anymore if a new intermediate protocol is added to the database, as sometimes it may require an update of the values assigned to protocols already in the database.

To simplify this task of assigning layer numbers to protocols, we propose an automated method, based on the PEG. Before going into details, it is worth noting that this procedure is just an heuristic and has no theoretical bases. Its only goal is to provide a strict ordering based on network protocols, because, to recognize tunnels, the exact layer associated with each protocol is not important *per se*, but only when it is compared with layers of other

protocols. For example, to identify a tunneled IP, it is necessary that both IP and IPv6 belong to the same layer, but it is not important what their actual layer value is: it could be different from level 3, as long as it is the same for both protocols.

The heuristic acts as follows: *(i)* the layer values for all nodes in the graph are set to INF (infinite), except for `startproto`, which gets the value 1; *(ii)* the recursive procedure defined in Algorithm 1 is called on the graph, starting from the node representing Startproto.

Algorithm 1 Assigning layer numbers to protocols

```
1: Procedure AssignProtoLevels (node n)
2:
3: if n.Visited then
4:    return
5: end if
6:
7: node.Visited = true
8: minSuccessor = GetMinSuccessor(n);
9: nextLevel = (minSuccessor ? minSuccessor.Level : INF)
10:
11: if nextLevel ≤ n.Level then
12:    maxPredecessor = GetMaxPredecessor(n)
13:    prevLevel = maxPredecessor.Level
14:    if prevLevel < nextLevel then
15:       n.Level = prevLevel + ((nextLevel-prevLevel)/2)
16:    end if
17: end if
18:
19: level = ceil(n.Level+1)
20: for all s ∈ n.successors do
21:    if level < s.Level then
22:       s.Level = level
23:    end if
24: end for
25:
26: for all s ∈ node.successors do
27:    AssignProtoLevels(n)
28: end for
```

Method `GetMinSuccessor` returns the smaller layer among those of a protocol's successors, while `GetMaxPredecessor` returns the greater layer among those of a protocol's predecessors. In both cases, self-loops are not considered.

The heuristic is best explained with an example. Figure 3.16 (a) depicts a simple PEG in which protocols are not associated with any layer (except Startproto, which by default gets the value 1). First, the procedure assigns to the successors of Startproto, in

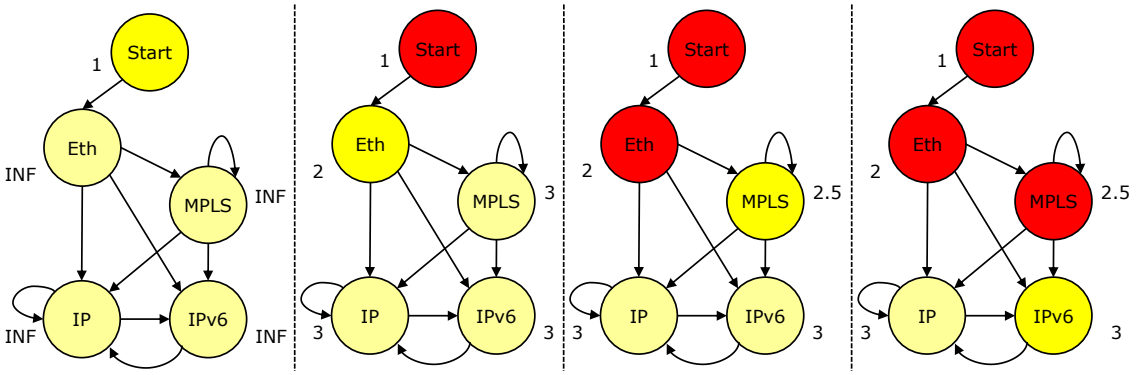


Figure 3.16. Layer assignment example.

this case Ethernet, the value `ceil(layer(startproto)+1)`, i.e. 2. The procedure is then repeated for Ethernet: all its successors (MPLS, IP, IPv6) get the value 3 (see Figure 3.16 (b)). When the procedure visits the node related to MPLS, it notices that the node's layer, which is 3, is equal to the lower layer among those of its successors. Therefore, the layer value for MPLS is updated to `prevlevel+((nextlevel-prevlevel)/2)`, i.e. 2.5 (see Figure 3.16 (c)). Because of the check in line 21 of Algorithm 1, the successors of MPLS are not updated. Finally, IP (Figure 3.16 (d)) and IPv6 are considered. Both are updated according with lines 14 and 15 of the pseudocode. However, since `prevlevel = nextlevel = 3` for both of them, their layers remain unchanged.

3.5.2 Building the xpFSA

In order to create an xpFSA identifying packets having some protocols encapsulated in a tunnel, a method very similar to the one already defined for header indexing is used.

In fact, as a first step, the skeleton of the automaton referring to the regular expression representing the header chain is built: the requirement that one or more protocols must be tunneled is ignored, because this information is not coded into the regular expression itself. As a consequence, no counters are defined yet.

At this point, for each `tunneled` keyword specified in the header chain and referring to a protocol of a different layer, a new counter is created. Its name is in the form `Ln.cntr`, where n is a number representing the network layer assigned to the protocol that must be tunneled. Furthermore, an operation of increment of this variable, i.e. `Ln.cntr++`, is associated with all symbols representing an encapsulation rule having as target a protocol belonging to a layer greater or equal to n.

During the labeling of the transitions, the one leading to a state associated with an element of the header chain on which the keyword `tunneled` has been specified, is associated with the evaluation of a predicate. In particular, this p-transition must verify if the proper counter has a value greater than one. For example, if a protocol that must be tunneled belongs to the layer 3, the p-transition leading to its related state tests the value

of the variable `L3.cntr`.

As an example, consider the non-deterministic automaton representing the header chain `ip tunneled`, shown in Figure 3.17 and built referring to the PEG of Figure 3.16. Since IP has been classified as belonging to the layer 3, the variable `L3.cntr` has been defined. This counter is incremented each time a symbol leading to IP or IPv6 is received, since they are the protocols defined in the PEG that have been assigned a layer greater than, or equal to 3. Moreover, the p-transition towards the right-most state, representing the IP header involved in a tunnel, fires only if the counter is greater than one, i.e. if at least another header belonging to a layer greater or equal to 3 has already been encountered within the packet under analysis[7].

Symbol	Operation(s)
eth-ip	L3.cntr++
eth-ipv6	L3.cntr++
mpls-ip	L3.cntr++
mpls-ipv6	L3.cntr++
ip-ip	L3.cntr++
ip-ipv6	L3.cntr++
ipv6-ip	L3.cntr++

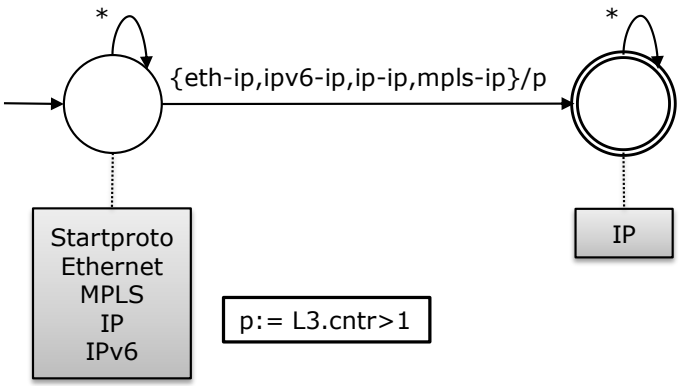


Figure 3.17. Non-deterministic automaton representing the filter `ip tunneled`.

Then, the automaton must be converted into a deterministic one, through the application of the rules defined in the xpFSA model, and then translated into the xpFSA recognizing packets matching the NetPFL header chain, with the application of the rules explained in Section 3.4.6 and following.

To conclude, it is important to highlight that on the same element of the header chain, all three constraints requiring p-transitions (i.e. the header indexing, predicates on protocol fields and the keyword `tunneled`) could be specified. If this happens, the p-transition leading to the state built from this element must evaluate all the conditions, which are joined using the Boolean operator `and`.

3.6 Using the xpFSA model in field extraction

This section presents an algorithm to build the xpFSA representing a NetPFL rule, if a field extraction was specified in the filter `Action`.

[7]It is worth remembering that the counter is increment as soon as the symbol is received, hence before the predicate evaluates its value.

Take for example `ip extractfields(ip%2.src)`: a packet, in order to match the filter, should have at least one IP header. Moreover, if a further IP exists, the NetPFL expression returns to the user application only the source address of this second instance of the protocol, instead of the entire packet. As a consequence, it requires that the analysis of the packet does not end when a first IP is reached, but continues until the eventual second one, in order to extract the required address value. Unfortunately, as shown in Figure 3.18, the xpFSA created using the algorithm of Section 3.4: *(i)* terminates the analysis as soon as an IP is found; *(ii)* does not have an action state associated with a second IP and extracting the value of its source address.

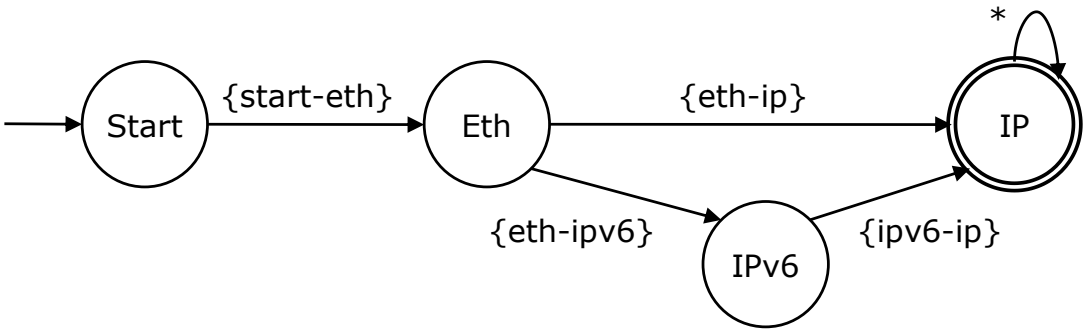


Figure 3.18. xpFSA matching the header chain `ip`.

Then, after the construction of the xpFSA representing the filtering conditions, further operations are required. In particular, each NetPFL element specified within the keyword `extractfields(...)` is considered and translated into a different xpFSA. To do this, it is used an algorithm very similar to the one already presented, and which is easily described referring to an action such as `extractfiedls(ip%2.src)`. First of all, `ip` is translated into the regular expression $.* \texttt{ip} .*$, as if it was a NetPFL header chain, and then converted into the non-deterministic automaton depicted in Figure 3.19. As highlighted with the dashed circle, the state coming from the element `ip` is an action state extracting the field `src`, and not an accepting state, as it would happen in case of an automaton modeling a filtering condition. Moreover, the last $.*$ of the regular expression, instead of originating a self loop on the rightmost state (firing on any received symbol), is modeled by creating a new state and by adding two transitions. The first transition connects the action state to this new state, and the other one is a self-loop on the state just created; both transitions fire on all symbols defined in the PEG in use. This way, the extraction is limited only to the protocol specified in the NetPFL expression.

Now, the non deterministic automaton just built is translated into a deterministic one by using the rules defined in the xpFSA model, and then converted into the xpFSA of Figure 3.20, through the labeling of the states representing the reaching of a single protocol, and the expansion of those associated with multiple protocols.

After their creation, both xpFSA (i.e. the one representing the filtering conditions and

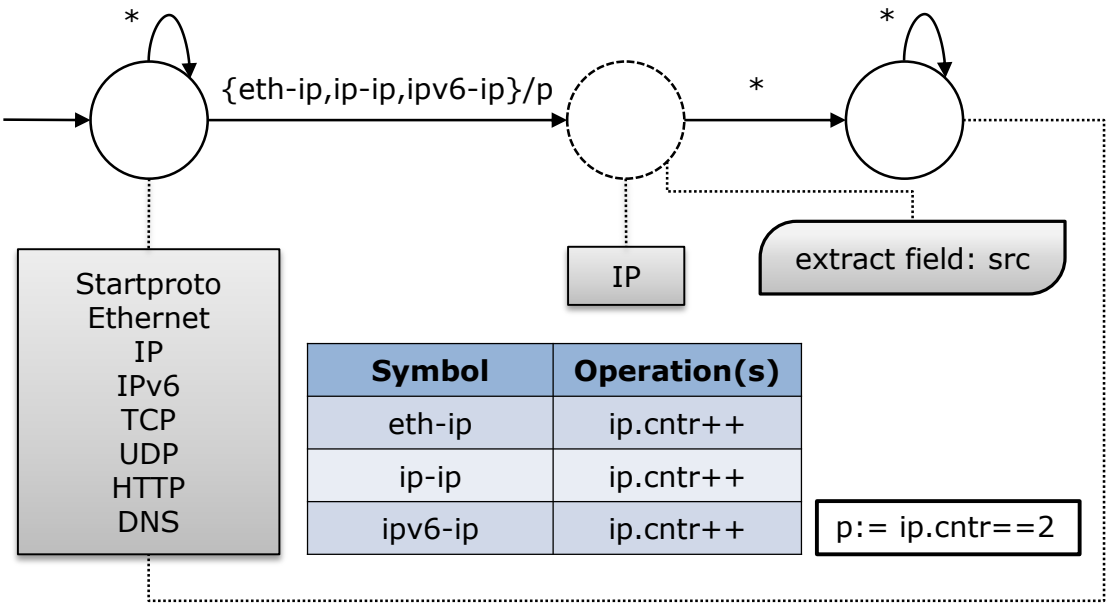


Figure 3.19. Non-deterministic automaton representing the extraction of `ip%2.src`.

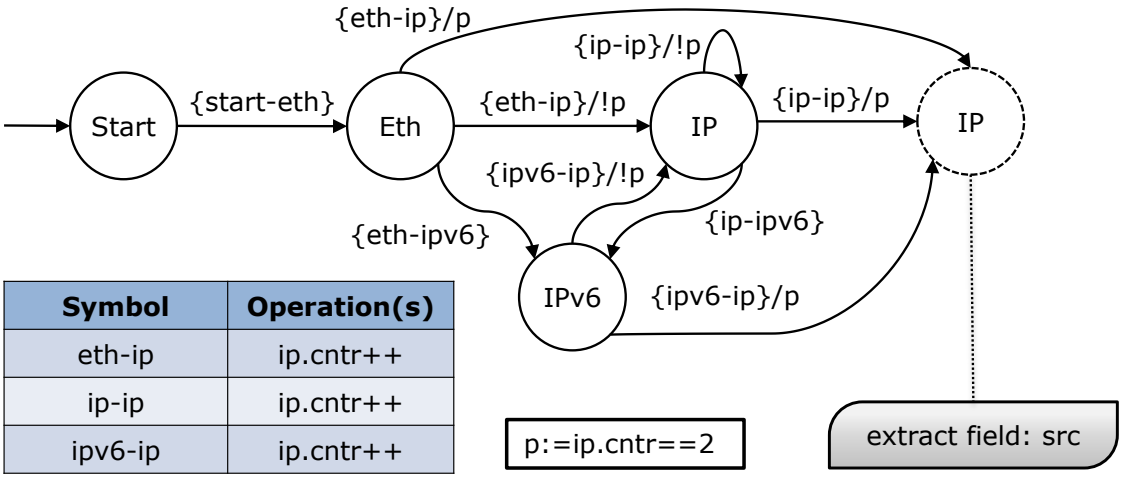
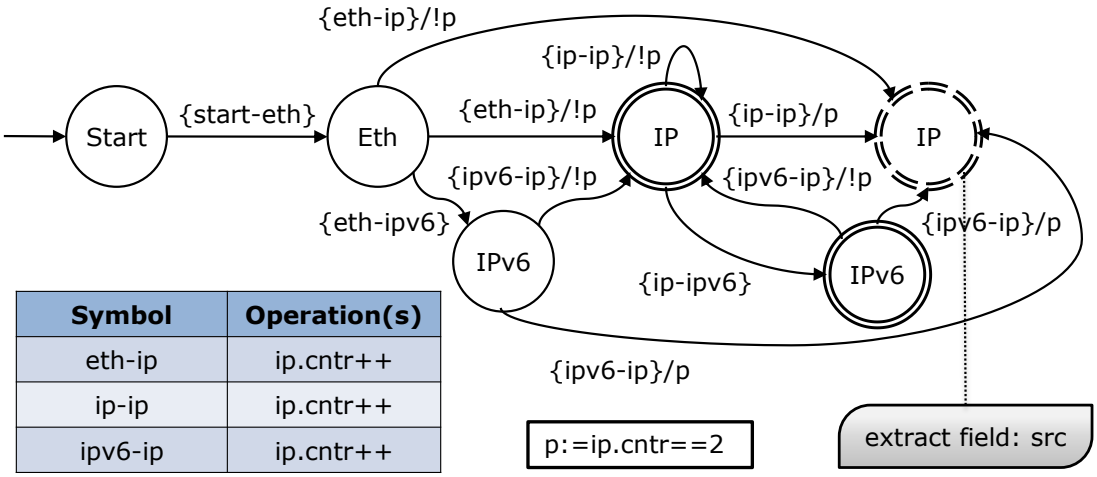


Figure 3.20. xpFSA representing the extraction of `ip%2.src`.

the one modeling the extraction action) are joined together using the Boolean operator `or`, in order to obtain a single automaton modeling the whole NetPFL rule. Figure 3.21, for instance, shows the xpFSA created from the NetPFL statement `ip extractfields(ip%2.src)`,

and obtained through the union of the automata of Figure 3.18 and 3.20, respectively representing the header chain and the action. As highlighted by the different notation used to represent it, the right-most state associated with IP in Figure 3.21, as well as being an accepting state, is also an action state extracting the value of the source address. Moreover, it is worth noting that, after the reaching of the left-most state representing IP, the packet is certainly accepted, since the filtering condition has already been verified.



Symbol	Operation(s)
eth-ip	ip.cntr++
ip-ip	ip.cntr++
ipv6-ip	ip.cntr++

Figure 3.21. xpFSA representing a NetPFL rule with field extraction.

3.6.1 An optimization

The just depicted algorithm can be used to model the xpFSA representing either kinds of extraction, i.e. `proto%n.field` and `proto*.field`. However this algorithm, as it has been defined, is not actually used if the NetPFL action requires to extract from the first instance of a protocol within the packet. For this (probably more common) case, we in fact implemented an optimization that avoids to check a counter to decide if the action state must be reached.

To understand this improvement, which affects the construction of the first non deterministic automaton representing the filter `Action`, consider the NetPFL rule `ip extractfields(tcp.sport)`[8]. As it is evident from Figure 3.22 (a), the self loop on the left-most state of this automaton does not fire with any symbol defined in the PEG, but with any symbol except those leading to the action state. This way, this state is reached only when the first TCP header is encountered within the packet, making useless the use of a counter associated with TCP. In Figure 3.22 (b), the complete automaton modeling the action is shown, while 3.22 (c) depicts the final xpFSA built from the NetPFL rule `ip`

[8]It is worth remembering that, in the extraction, `tcp.sport` has the same meaning of `tcp%1.sport`.

`extractfields(tcp.sport)`. It is worth noting that this automaton has been obtained joining the one of Figure 3.18 (representing the filtering condition `ip`) with the xpFSA of Figure 3.22 (b), using the Boolean operator `or`.

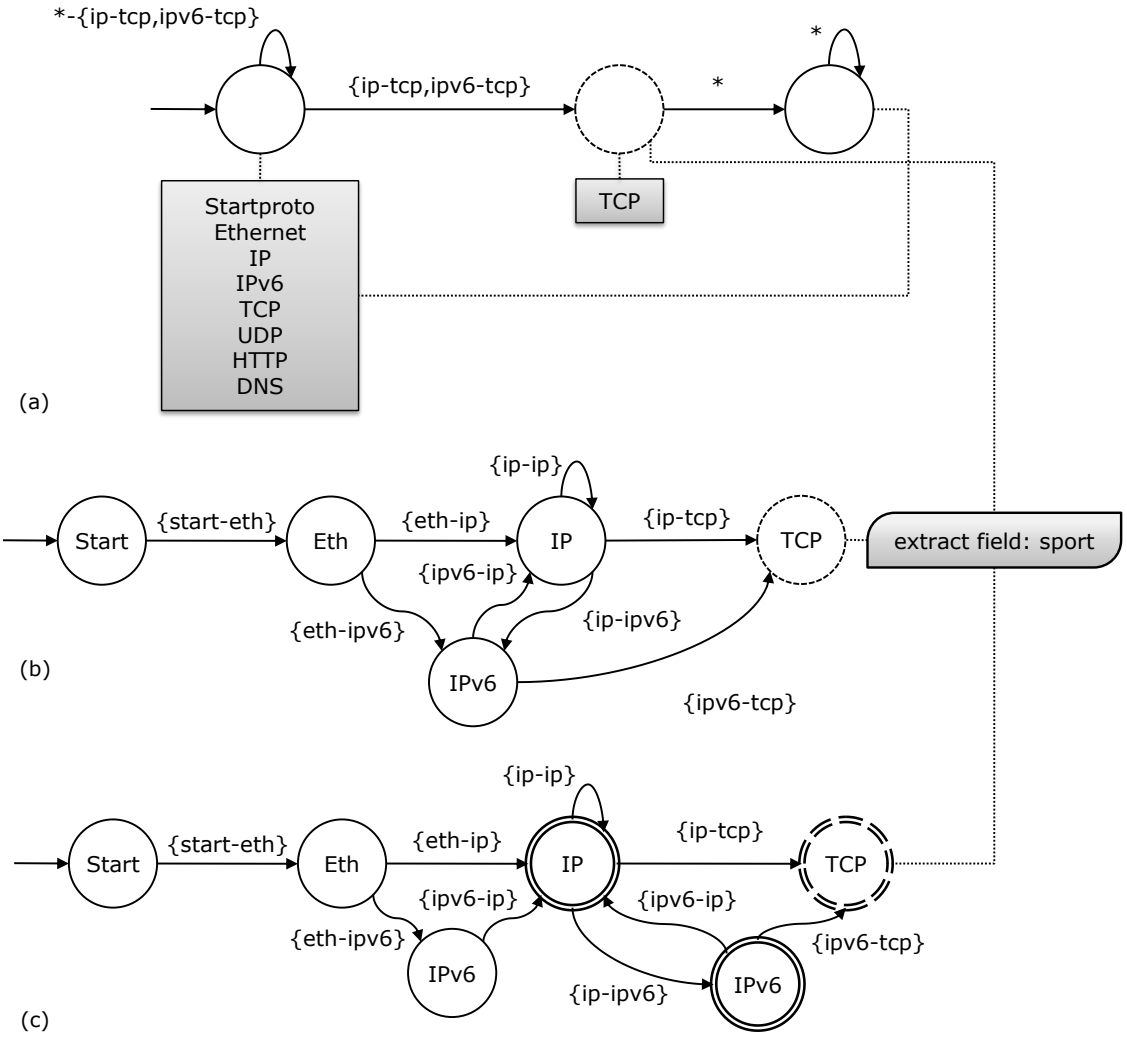


Figure 3.22. Building the xpFSA to extract from the first TCP header of the packet.

3.7 The preferred encapsulation rules

As explained above, the NetPFL statements are translated into xpFSA, by taking into account the protocol encapsulation rules represented in a PEG. The time needed to create the automaton increases with the size of the PEG, which influences both the number of states, and the number of transitions exiting from each state. Furthermore, having

more states causes the generation of more executable code, while the number of outgoing transitions influences the time needed for the code to determine which is the next protocol within the packet. As a consequence, small PEGs produce reduced xpFSA and more powerful filtering programs.

By default, the NetPFL language refers to the PEG built considering only the *preferred* encapsulation rules defined in the given database. Instead, if the full database should be considered, the filtering statement must include the `fullencap` keyword, so that the *Full PEG* is used while building the xpFSA.

This section presents our improvement to the Network Packet Description Language [28] (NetPDL), which enables the user to indicate which are his preferred encapsulation rules.

In detail, we defined a new Boolean attribute called, unsurprisingly, *preferred*, which can be specified in both the `<nextproto>` and `<nextproto-candidate>` elements, i.e. those elements used to define the protocol encapsulation rules. This way, when the keyword `fullencap` is not provided in the NetPFL statement, only the rules having `preferred=''true''` are considered; otherwise, each rule in the database is taken into account.

As an example, consider the NetPDL excerpt shown in Figure 3.23: if the keyword `fullencap` has been specified, it results in the fragment of PEG in the left of Figure 3.24; otherwise it is represented by the PEG shown in the right of the same picture.

```
<protocol  name="ip">
  <encapsulation>
    <switch expr="buf2int(nextp)">
      <case value="4"> <nextproto proto="#ip"/> </case>
      <case value="6"> <nextproto proto="#tcp" preferred="true"/> </case>
      <case value="17"> <nextproto proto="#udp" preferred="true"/> </case>
      <case value="41"> <nextproto proto="#ipv6"/> </case>
    </switch>
  </encapsulation>
</protocol>
```

Figure 3.23. Example of NetPDL encapsulation rules.

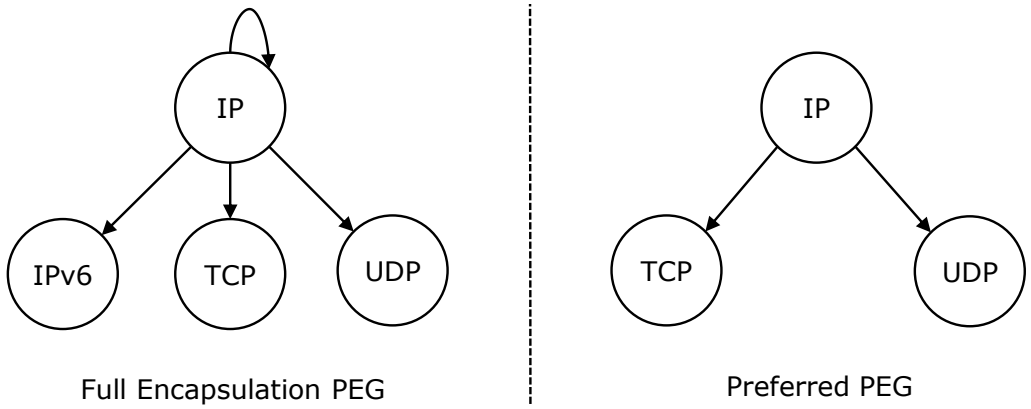


Figure 3.24. Two fragments of PEG.

3.8 Implementation

The proposed model has been implemented in the NetBee[9] library, which features an experimental compiler that creates run-time code for the NetVM [15] virtual machine. The front-end compiler [16] takes the filtering expression expressed as a NetPFL [29] string and a NetPDL [28] protocol database to generate an in-memory representation of the xpFSA filter. This code is then translated into NetIL code, a NetVM-specific assembly-like language. The generated code can be executed in a NetVM interpreter, or compiled Just-In-Time (JIT) if a backend compiler is available for the target architecture.

The implementation of the xpFSA model presented the same challenges of its predecessor, pFSA. Please refer to the detailed description given in Section 2.6 for more information.

Since the previous implementation was very modular already, adding the new model features was not complicated: handling of counters and of actions states was not difficult. Moreover, the implementation of the new algorithms was simple because of partial code reuse from the pFSA model.

3.9 Validation

The xpFSA model was validated through three categories of tests, aimed at evaluating different aspects of the formalism: *(i)* compile-time performance, *(ii)* run-time performance and *(iii)* impact of counters on the number of states. Tests were executed on a workstation equipped with an Intel E8400 Core 2 Duo dual-core processor with 12GiB of RAM, running a 64-bit version of Ubuntu Linux 10.04 (kernel 2.6.32-45-generic). All test processes were bound to a single processor, with hot disk and processor caches, and the machine

[9] `http://www.nbee.org`

was otherwise unloaded. Time measurements were performed using the `gettimeofday()` UNIX function.

At the best of the authors' knowledge, no other existing filtering language allows users to specify filters with encapsulation constraints and actions to be executed when the filter is satisfied. For example, neither `libpcap` [8], representing the foundation of many packet filtering tools (e.g., `tcpdump`, Wireshark), nor the display filters implemented in Wireshark [3] (which replace the basic filtering capabilities of libpcap when packets have to be shown on screen) support filtering based on protocol encapsulation rules. As a consequence, we could not compare the performance of our implementation with other competitors. However, we took care of obtaining our results by using a framework that has already been proven to be at least equivalent to the state of the art in this field [16].

3.9.1 Compilation time

This test evaluates the filter compilation time, i.e., the time required for generating the actual x64 assembly code implementing a specific NetPFL filtering expression. This process includes an initial step represented by our algorithm followed by a very complex compilation and optimization process (part of the NetVM framework) before the final code emission. The sample NetPFL filters are shown in Table 3.2, while the PEG of reference is depicted in Figure 3.25, as well as the network layer associated with each protocol[10].

#	**Filtering expression**
1	tcp
2	tcp in ip
3	tcp in ip in ethernet
4	tcp in ip in ethernet in startproto
5	tcp in ip in ppp in gre in ip
6	tcp in ip in ppp in gre in ip in ethernet
7	tcp in ip in ppp in gre in ip in ethernet in startproto
8	tcp in ip notin ethernet
9	tcp in ip%2
10	tcp in ip tunneled

Table 3.2. Sample NetPFL filters.

In order to measure the time spent by our algorithm with respect to the total code generation time, each filter was compiled thousand times and averaged. This way, we

[10]Note that, for the sake of clarity, this PEG is a reduced version of that available at the nbee.org website, which accounts for more than 100 protocols.

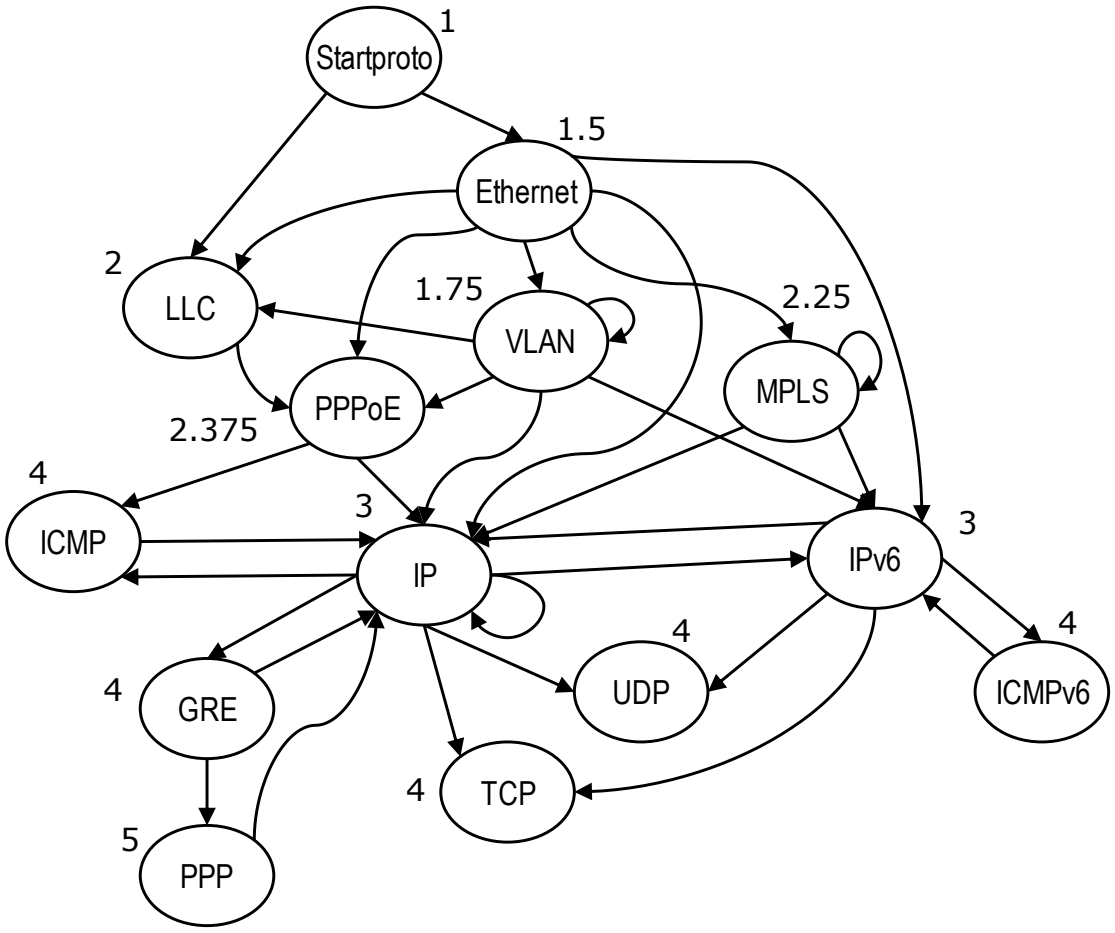


Figure 3.25. Protocol Encapsulation Graph.

obtained the numbers depicted in Figure 3.26, where the first two series represent the computation time obtained when using the default `returnpacket` action (i.e., no field extraction is performed) and the last two represent the computation time measured when the `extractfields(tcp.sport)` action is also included in the filtering expression.

Results show that the time required for creating the xpFSA is one order of magnitude less than the total generation time, for almost all the considered filters. In particular, as is evident by comparing expressions from `#1` to `#4`, the total time decreases by increasing the precision of the filter. On the other hand, the generation of the xpFSA is faster if there are less protocols that could match the first ".∗" element of the regular expression associated with the filter. In fact, our algorithm first represents this element with a state that will be expanded in most of the protocols defined in the PEG, and then it prunes the unnecessary ones. As a consequence, NetPFL filters that explicitly mention `startproto` (such as `#4` and `#7`) generate very compact automata, and represent the fastest generation

case for our algorithm.

Figure 3.26 also shows how the field extraction introduces a negligible overhead in almost all the sample filters, especially with regard to the total compilation time. It is worth pointing out that this overhead is mainly caused by the generation of instructions that actually perform the extraction, i.e., that store fields in a specific memory segment of the NetVM process. Instead, the creation of the automaton representing the whole filtering expression requires more time because two xpFSA are created (one representing the filter, and one modeling the action) and combined together using the Boolean operation `or`; the resulting xpFSA is then determinized and minimized.

As a final remark, it is worth highlighting that we are able to reduce the overall filter compilation time, and then improve the compile-time performance of the system, by increasing the time spent in the first compilation stage, i.e. when our algorithm creates the xpFSA.

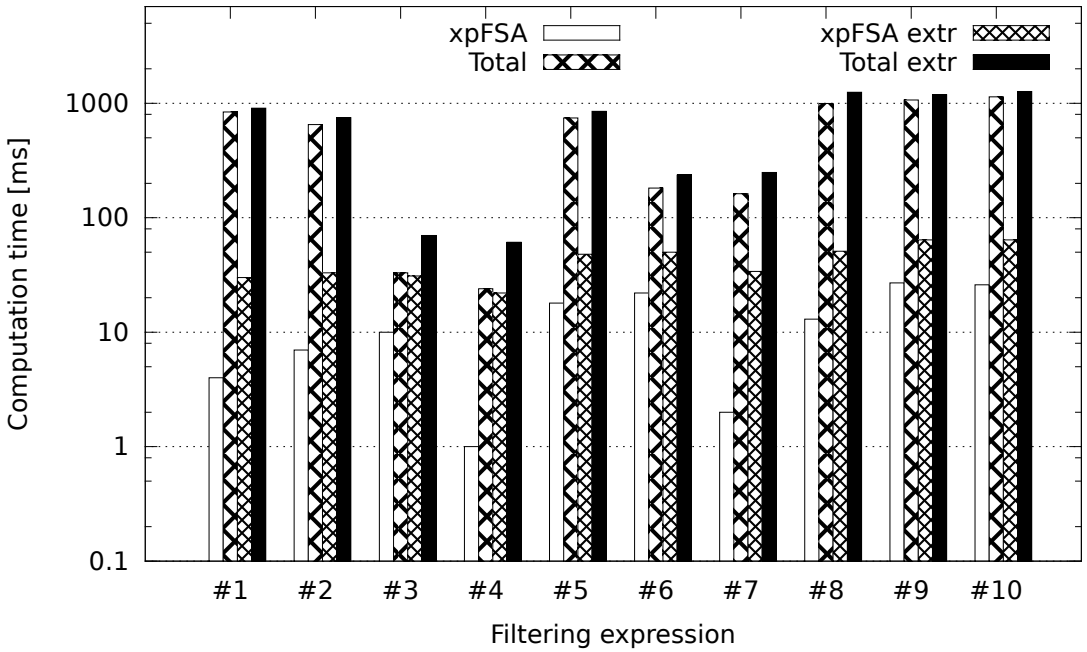


Figure 3.26. Performance of the code generator.

3.9.2 Filtering time

This test aims at evaluating the quality of the resulting filtering code, i.e., the x64 assembly program that actually analyzes packets. We executed filters of Table 3.2 on a synthetic trace composed of four packets repeated as many times as needed in order to obtain 1 GiB of traffic. The packets we used aimed at reproducing the most common encapsulations on

the Internet, i.e., `ethernet-ip-tcp`, `ethernet-ip-udp`, `ethernet-ip-gre-ppp-tcp` and `ethernet-ip-gre-ppp-udp`[11]. The results we obtained are depicted in Figure 3.27.

As it is evident, the number of packets analysed each second increases when the filter is more specific, i.e. leaves less freedom to the protocols that may appear in a certain position of the packet.

Moreover, the field extraction does not greatly reduce the performance of the filter.

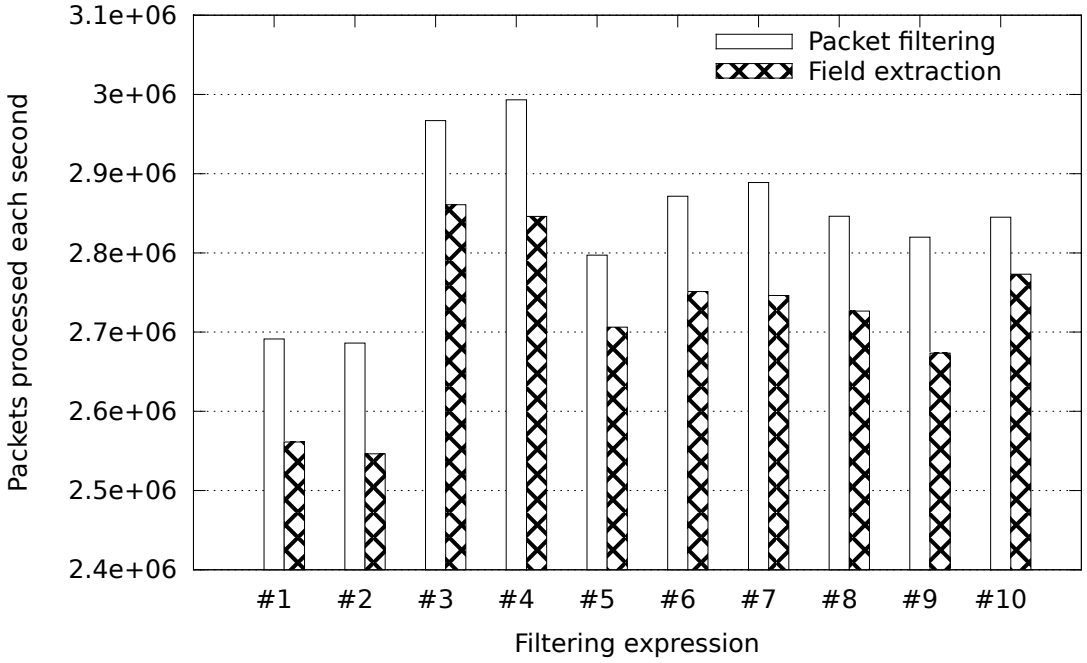


Figure 3.27. Performance of the generated filters.

3.9.3 Impact of counters

The next test points out how an xpFSA could have a reduced number of states, if compared with a pFSA representing the same filtering expression. In particular, this happens when counters are used, i.e., in case of filters requiring at least `n` instances of a protocol (e.g., `vlan%2`), or requiring that a protocol is involved in a tunnel (e.g., `ipv6 tunneled`).

Figure 3.28 portraits this reduction by comparing the pFSA and the xpFSA associated with filters that require an increasing number of IP headers within valid packets. The trend of the number of states is shown both using the PEG depicted in Figure 3.25, called *full*, and a *core* PEG, which includes only the encapsulations: `Startproto` → `Ethernet`, `Ethernet` → `IP`, `Ethernet` → `IPv6`, `IPv6` → `IP`, `IP` → `IP` and `IP` → `IPv6`.

[11]It is worth noting that the last two packets are commonly encountered when using a VPN tunnel.

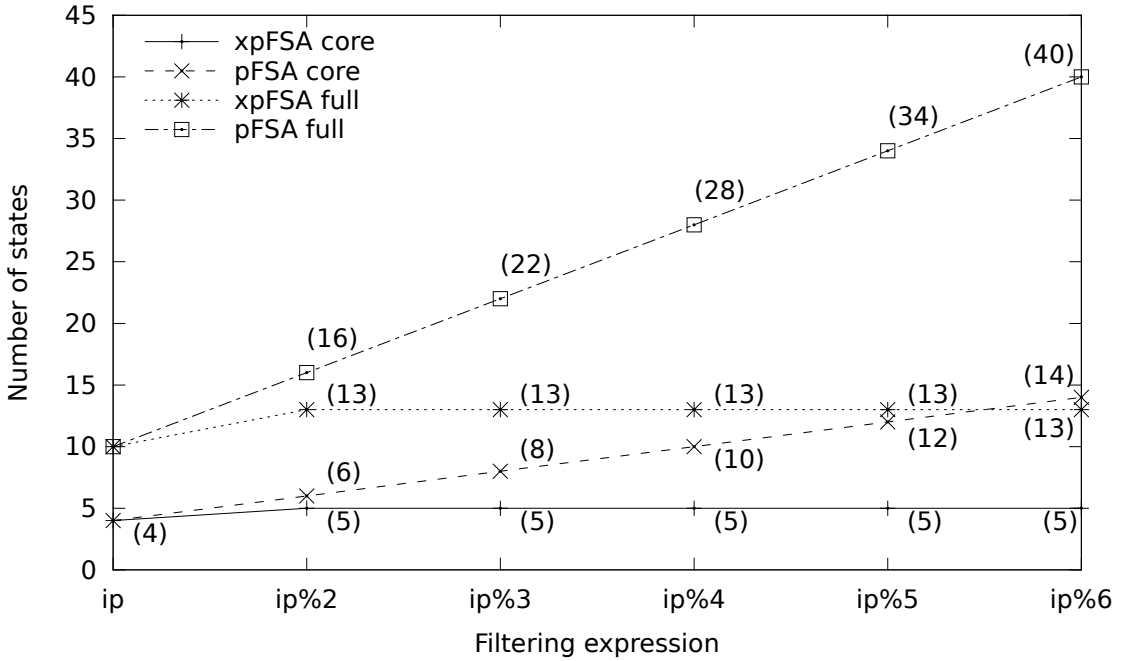


Figure 3.28. Difference in the number of states.

From the figure, it is immediately evident that, regardless of the PEG of reference, the number of states of the pFSA grows steadily with the number of required IP headers. As illustrated in Figure 3.29, this happens because the automaton representing `ip%n` consists in the pFSA associated with `ip%(n-1)`, enriched with all paths leading from IP to IP. The size of the *step*[12] depends then on the number of protocols that the PEG allows between IP and IP, as is evident by comparing the trends labeled with *pFSA core* and *pFSA full* in Figure 3.28.

With regard to the xpFSA, we can observe an increase in the number of states only between filters `ip` and `ip%2`, while, from `ip%2` onwards, the state count remains constant. Moreover, it is interesting to note how this step is reduced compared to that related to the pFSA representing the same expressions. This is because, if we consider for instance the *full* PEG, in our formalism filter `ip%2` only requires a new state associated with IP, and two states that represent respectively protocols GRE and PPP. Instead, the pFSA requires also to duplicate the states associated with IPv6, ICMPv6, and ICMP.

[12]With the word *step*, we mean the difference in the number of states of the automaton modeling the filter `proto%(n-1)`, and the one modeling the filter `proto%n`.

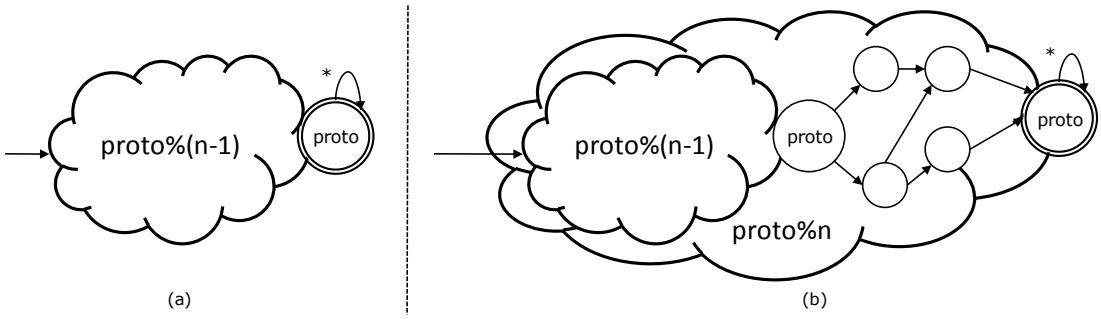


Figure 3.29. From the pFSA representing `proto%(n-1)`, to the pFSA modeling `proto%n`.

3.10 Conclusions

This chapter presented xpFSA, a model for packet filter that aims at improving performances in case of tunneled encapsulations. This feature is of critical importance, because we can observe an increasing complexity in the lowest layers of the protocol stack, arising in particular from network virtualization. It is important to be able to capture all the traffic we are interested in (e.g., web traffic), independently from the actual encapsulations used at lower layers. Furthermore, being agnostic with respect to network protocols, our implementation exploits a dynamic protocol database that allows to change the protocols it operates upon by simply updating those files at run-time, without having to modify the source code of the packet filter compiler itself.

The performance of our model could not be compared against the one of our competitors, because none supports filtering based on protocol encapsulation rules. However, we took care of obtaining our results by using a framework that has already been proven to be at least equivalent to the state of the art in this field. On the other hand, it was proven that the action of field extraction does not reduce the performance of the filter. In the end, the improvements w.r.t. the previous pFSA model, measured in the number of states in the generated automaton, have been demonstrated.

Conclusions

This dissertation introduced a number of changes in the domain of packet filters. The present and future challenges related to packet filters were discussed, and multiple solutions were presented.

First, the flexibility problem was addressed. No packet filtering language currently supports natively the specification of rules based on a specific protocol encapsulation. Furthermore, it is often very difficult to support tunneled protocols. This dissertation introduced NetPFL, a new declarative language for data-plane packet processing, whose main strengths are:

- it can handle complex situations of tunneled and stacked encapsulations, giving the user a finer tuned control over the semantics of a filtering expression;
- it supports several independent filters that can lead to multiple matches;
- it allows the user to choose, in an implementation-agnostic way, the action that should be performed upon receipt of each matching packet;
- it associates each packet with a stream indicator, to facilitate the merging of different logical filters within the same physical filtering machine and the demultiplexing of the associated packets that belong to different logical filters;
- it is human-friendly, making it suitable for fast command-line processing.

Consequently, the efficiency of the current state of the art of packet filters was analyzed, and the most common flaws were exposed, while filtering packets with bizarre or unusual network encapsulations. A new model of packet filters, called pFSA, was outlined, that ensures the optimal number of checks on the packet in order to take the matching/not-matching decision. This result is obtained by transforming packet filtering rules into Finite State Automata (FSA), which guarantee optimal results even in case of multiple filters combined together. This was achieved by augmenting the transitions inside the automaton with Boolean predicates, that are modeled as hypotheses on specific properties of the protocols included in the packet itself. The model was proven to be as fast as the best competitors for simple packet filters and to scale linearly with the number of predicates on the same protocol, such as when filtering multiple TCP sessions. At the same time it demands limited processing and memory requirements in the filtering code generation phase, which represents a huge improvement when compared with other approaches.

In the end, the model was further expanded in xpFSA, in order to improve the previous model by extending its support for tunneling features. This was achieved by associating the input symbols with operations on specific counters, and by allowing the transitions to be labeled with predicates expressed on the value of these counters. It was also proven that the extended formalism does add noticeable overhead to the system.

Bibliography

[1] F. Risso, L. Degioanni, An Architecture for High Performance Network Analysis. In *Proceedings of the 6th IEEE Symposium on Computers and Communications (ISCC 2001)*, Hammamet (Tunisia), pp. 686-693, July 2001.

[2] The PCAP Library Man Page. Available at `http://www.tcpdump.org/pcap3_man.html`

[3] G. Combos, The Wireshark Network Protocol Analyzer. Available at `http://www.wireshark.org/`

[4] R. Pang, V. Paxson, R. Sommer, L. Peterson, Binpac: a yacc for writing application protocol parsers. In *Proceedings of the 6th ACM Internet Measurement Conference*, pp. 289-300, Rio de Janeiro, Brazil, October 2006.

[5] M Roesch, Snort - Lightweight Intrusion Detection for Networks. In *Proceedings of the 13th Systems Administration Conference (LISA '99)*, pp. 229-238, Seattle, WA, November 1999.

[6] V. Paxson, Bro: A System for Detecting Network Intruders in Real-Time, *Computer Networks*, Vol. 31, No. 23-24, pp. 2435-2463, Elsevier, December 1999.

[7] StreamBase Systems, StreamSQL online documentation. Available at `http://streambase.com/developers/docs/latest/streamsql/index.html`, 2007.

[8] S. McCanne, V. Jacobson, The BSD Packet Filter: A new architecture for user-level packet capture. In *Proceedings of the 1993 Winter USENIX Technical Conference*, San Diego, CA, pp. 259-269, Jan. 1993.

[9] M.L. Bayley, B. Gopal, M.A. Pagels, L.L. Peterson, PATHFINDER: A pattern-based packet classifier. In *Proceedings of the First USENIX Symposium in Operating System Design and Implementation*, Monterey, CA, pp. 115-123, Nov. 1994.

[10] D.R. Engler, M.F. Kaashoek, DPF: Fast, flexible message demultiplexing using dynamic code generation. In *Proceedings of ACM SIGCOMM '96*, Stanford, CA, pp. 53-59, Aug. 1996.

[11] A. Begel, S. McCanne, S.L. Graham, BPF+: exploiting global data-flow optimization in a generalized packet filter architecture. In *SIGCOMM Computer Communication Review*, Vol. 29(4), pp. 123-134, Oct. 1999.

[12] Z. Wu, M. Xie, H. Wang, Swift: a fast dynamic packet filter. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, San Francisco, CA, pp. 279-292, Apr. 2008.

[13] P. Rolando, R. Sisto, F. Risso, SPAF: stateless FSA-based packet filters. In *IEEE/ACM Transactions on Networking (TON)*, Volume 19 Issue 1, Feb. 2011.

[14] H. Bos, M. Cristea, T. Nguyen, G. Portokalidis, FFPF: Fairly Fast Packet Filters. In *Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation (OSDI04)*, San Francisco, CA, pp. 347–363, Dec. 2004.

[15] O. Morandi, F. Risso, P. Rolando, S. Valenti, P. Veglia, Creating Portable and Efficient Packet Processing Applications. In *Springer Design Automation for Embedded Systems*, Vol. 15, No. 1, pp. 51-85, March 2011.

[16] O. Morandi, F. Risso, M. Baldi, A. Baldini, Enabling Flexible Packet Filtering Through Dynamic Code Generation. In *Proceedings of IEEE International Conference on Communications (ICC 2008)*, Beijing, China, pp. 5849-5856, May 2008.

[17] O. Morandi, F. Risso, S. Valenti, P. Veglia, Design and Implementation of a Framework for Creating Portable and Efficient Packet Processing Applications. In *Proceedings of the 7th ACM International Conference on Embedded Software (EMSOFT 2008)*, Atlanta, GA, pp. 237-244, Oct. 2008.

[18] O. Morandi, F. Risso, P. Rolando, O. Hagsand, P. Ekdahl, Mapping Packet Processing Applications on a Systolic Array Network Processor. In *IEEE International Workshop on High Performance Switching and Routing (HPSR 2008)*, Shanghai, China, pp. 213-220, May 2008.

[19] L. Degioanni, M. Baldi, F. Risso, G. Varenni, Profiling and optimization of software-based network-analysis applications. In *Proceedings of the 15th Symposium on Computer Architecture and High Performance Computing*, Washington, DC, USA, p. 226, 2003.

[20] George Varghese. *Network algorithmics: an interdisciplinary approach to designing fast networked devices*. Morgan Kaufmann, 2005.

[21] J.E. Hopcroft, R. Motwani, J.D. Ullman. *Automata Theory, Languages, and Computation*. Addison-Wesley, 3rd Edition, 2006.

[22] I. Cerrato, M. Leogrande, F. Risso, Filtering Network Traffic Based on Protocol Encapsulation Rules. In *Proceedings of the International Conference on Computing, Networking and Communications (ICNC 2013)*, San Diego, CA, Jan. 2013.

[23] J.C. Mogul, R.F. Rashid, M.J. Accetta, The packet filter: An efficient mechanism for user-level network code. In *Proceedings of 11th ACM Symposium on Operating Systems Principles*, Austin, TX, pp. 39-51, Nov. 1987.

[24] T. Hruby, K. van Reeuwijk, H. Bos, Ruler: High-Speed Packet Matching and Rewriting on NPUs. In *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems (ANCS '07)*, Orlando, FL, pp. 1–10, Dec. 2007.

[25] G. Van Noord, D. Gerdemann, Finite state transducers with predicates and identities. In *Grammars*, Vol. 4, No. 3, pp. 263-286, 2001.

[26] R. Sekar, P. Uppuluri, Synthesizing fast intrusion prevention/detection systems from high-level specifications. In *Proceedings of the 8th conference on USENIX Security Symposium*, Vol. 8, pp. 6, 1999.

[27] R. Smith, C. Estan, S. Jha, I. Siahaan, Fast Signature Matching Using Extended Finite Automaton (XFA). In *Proceedings of the 4th International Conference on Information Systems Security (ICISS 2008)*, Hyderabad, India, pp. 158-172, December 2008.

[28] F. Risso, M. Baldi, NetPDL: an extensible XML-based language for packet header description. In *Comput. Netw.*, Vol. 50, No. 5, pp. 688–706, 2006.

[29] L. Ciminiera, M. Leogrande, J. Liu, O. Morandi, F. Risso, A Tunnel-aware Language for Network Packet Filtering. In *Proceedings of the 2010 IEEE Global Telecommunications Conference (GLOBECOM 2010)*, Miami, FL, pp. 1–6, Dec. 2010.

[30] NetBee, a powerful library for generic packet processing. Available at `http://nbee.org/`.