Enabling precise traffic filtering based on protocol encapsulation rules

Enabling Precise Traffic Filtering Based on Protocol Encapsulation Rules

Ivano Cerrato[a,*], Fulvio Risso[a]

[a]*Department of Control and Computer Engineering, Politecnico di Torino, Italy*

Abstract

Current packet filters have a limited support for expressions based on *protocol encapsulation relationships* and some constraints are not supported at all, such as the value of the IP source address in the inner header of an IP-in-IP packet. This limitation may be critical for a wide range of packet filtering applications, as the number of possible encapsulations is steadily increasing and network operators cannot define exactly which packets they are interested in. This paper proposes a new formalism, called eXtended Finite State Automata with Predicates (*xpFSA*), that provides an efficient implementation of filtering expressions, supporting both constraints on protocol encapsulations and the composition of multiple filtering expressions. Furthermore, it defines a novel algorithm that can be used to automatically detect tunneled packets. Our algorithms are validated through a large set of tests assessing both the performance of the filtering generation process and the efficiency of the actual packet filtering code when dealing with real network packets.

Keywords: Packet Filtering, Protocol Encapsulations, Protocol Encapsulation Constraints, Construction Algorithm, Augmented Finite State Automata, xpFSA, NetPFL

*Corresponding author. Email address: ivano.cerrato@polito.it.

1. Introduction

While protocol encapsulations were rather simple in the past (e.g., TCP/UDP in IP in Ethernet), new necessities, arising in particular from network virtualization, are rapidly increasing the complexity of the protocol stack. This impacts on the complexity of packet filters, which represent the basic building blocks for many applications such as firewalls and network monitors. In fact, while on the one hand packet filters should be able to capture all the traffic of interest (e.g., web traffic) independently from the actual encapsulations used at the lower layers (e.g., plain Ethernet or a tunnel transporting IPv6 traffic over IPv4 networks), on the other hand they should allow to finely select/filter only packets that include specific protocol encapsulations (e.g., PPP in GRE, TCP in the second IP header instance of the packet).

Traditional packet filters, which are based on the existence of some protocol and on the value of some protocol fields, do not allow such a precise selection of traffic according to the encapsulations found in packets. For example, they cannot specify the value of the IP source address in the inner header of an IP-in-IP packet.

The precise filtering of such traffic requires both a packet filtering language that allows to express conditions on the encapsulation relationships between protocols, and an efficient implementation of that language in order to cope with the speed of current networks. While the Network Packet Filtering Language (NetPFL) [1] already addresses the first point, its implementation is still partial and not optimized in case of complex protocol encapsulation rules [2].

Based on the above considerations, this paper brings the following contributions to packet filtering. First, it proposes the *eXtended Finite State Automata with Predicates (xpFSA)*, a new formalism to represent filtering expressions and that extends the pFSA (Finite State Automata with Predicates) packet filtering model [3]. Like its ancestor, xpFSA guarantees the optimal number of checks on packet fields in order to identify their possible match of the filtering expression, even in case of composition of multiple filters. In addition, it introduces counters

and elementary operations that reduce the number of states of the automaton, which results in a more efficient generation of the executable code implementing the packet filter. Second, the paper defines an algorithm that transforms filtering expressions (potentially including complex protocol encapsulation constraints) into xpFSA, which completely replaces the automaton building process defined for pFSA and that cannot be used in case of complex encapsulation patterns. Third, it proposes a novel algorithm that can be used to automatically assign protocols to network layers, which is exploited to detect tunneled encapsulations.

In order to evaluate our algorithms and the filtering code generated from a xpFSA, we implemented them into the NetBee library [4]. Notably, our implementation does not require *a priori* protocols definition; in fact, it exploits a protocol database provided at run-time that can be easily extended or modified in order to recognize any new protocol and/or encapsulation, according to the properties of the NetPDL language [5]. In other words, our implementation can support both current and future protocols and encapsulations seamlessly, provided that the proper description is included in the protocol database.

This paper is structured as follows. Section 2 discusses the related works, while Section 3 summarizes the main characteristics of the NetPFL language and the pFSA packet filtering model. Section 4 presents the xpFSA model, while the algorithm to transform a NetPFL filtering expression into a xpFSA is detailed in Section 5. Section 6 shows the algorithm that automatically associates protocols to network layers. Section 7 provides an overview of the implementation; experimental results are then shown in Section 8, while Section 9 concludes the paper.

2. Related Work

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 protocol encapsulation constraints. For example, neither libpcap [6], repre-

senting the foundation of many packet filtering tools (e.g., `tcpdump`, Wireshark), nor the Wireshark display filters [7] (which replace the basic filtering capabilities of libpcap when packets have to be shown on screen) support such filtering expressions. Instead, the Network Packet Filtering Language (NetPFL) [1] supports protocol encapsulation patterns in the language definition, but its implementation is partial and limited to traditional packet filters with simple encapsulation rules [2].

Traditional packet filters, such as the CMU/Standford Packet Filter [8], the BPF [6] and BPF+ [9], PathFinder [10] and the Dynamic Packet Filter (DPF) [11], focus more on the filtering architecture (a.k.a., virtual machine), leaving less attention to the programming abstraction. Moreover, they do not support constraints on protocol encapsulation patterns and rely on ad hoc optimizations often inspired by compiler-oriented techniques, which are then applied to the code to be executed.

To support filtering expressions including protocol encapsulation constraints, this paper proposes xpFSA, namely an extension of the pFSA packet filtering model that enables to reuse optimal composition rules and optimization techniques defined in the automata theory [12]. In fact, the idea of extending an FSA is not new when looking at the broader field of packet processing; for instance, xpFSA takes some inspiring idea from the following proposals, although none of them was designed (nor able) to satisfy our objectives, some not being able even to filter packets.

The eXtended Finite Automata (XFA) [13] formalism augments traditional FSA with a finite memory and generic executable code to manipulate this memory, which is oriented to improve efficiency of signature matching in network intrusion detection systems. Similar ideas can be found also in the Extended Finite State Automata (EFSA) [14], which extends traditional FSA with finite sets of variables in order to model fast intrusion detection and prevention systems. However, its design goals are rather different, as EFSA is used to monitor sequences of system calls, which also requires a completely different algorithm to build the automaton. Similar considerations hold also for pfsr [15],

a predicate-augmented finite state recognizer that aims at simplifying the FSA used in natural language processing. Ruler [16] is a packet rewriter designed to anonimize traffic traces, which can also be used for packet filtering. It exploits a generalization of the FSA model called Tagged DFA [17] and uses variables to store the current position in the input string. FlowSifter [18] and COPY [19] extend context free grammars, regular grammars and automaton with predicates on transitions, variables and actions. Particularly, they define *Counting Regular Grammars* (CRG) [18] and *Distinguishable Counting Regular Grammars* (DCRG) [19] as extensions of regular grammars that use counters; albeit their theoretical degree of expressiveness is equivalent to our proposal, the implementation is rather different and targets a diverse use case. In fact, they aim at efficiently parsing application layer protocols (e.g., Facebook, Youtube) and extract fields of such protocols, while the goal of our work is to recognize packets satisfying constraints expressed on protocol encapsulations, with strong requirements in terms of real-time recomputation of the filtering code.

The Stateless FSA-based Packet Filter (SPAF) [20] model for packet filtering, which is the predecessor of pFSA and xpFSA, guarantees code optimality and safety, and it could be used to represent filtering expressions that include protocol encapsulation constraints. However, it is extremely slow in the automata generation phase because of the large number of generated states, and it is therefore suitable only for applications that can tolerate long filter generation time.

Finally, an early ancestor of the algorithm described in this paper has been presented in [2]; however, that algorithm does not support filters including the header indexing, the tunneling constraint and predicates on protocol fields (described in Section 3.1).

3. Background

3.1. Network Packet Filtering Language (NetPFL)

The NetPFL [1] is a declarative high-level language aimed at describing the conditions that a packet must satisfy in order to be accepted. Unlike other languages for packet filtering, NetPFL does not define any protocol header and encapsulation by itself, but it exploits definitions described externally, e.g., through the Network Packet Description Language (NetPDL) [5]. Moreover, NetPFL filtering expressions, or *header chains*, extend the traditional conditions based on the existence of some protocols and on the value of some protocol fields with conditions based on protocol encapsulation patterns, such as a specific chain of protocol headers.

This is achieved with the `in` and `notin` keywords, requiring respectively that, within a packet, the left-hand protocol is directly encapsulated into the right-hand one, or that the left-hand protocol is encapsulated in any protocol but the right-hand one. For instance, `tcp in {ip,ipv6}` matches packets having TCP directly encapsulated in IP or IPv6, while `tcp notin ip` accepts packets in which TCP is encapsulated in any protocol but IP. To define an encapsulation in which any protocol is valid, the literal `any` can be used; as an example, `tcp in any in ppp` is satisfied by packets having the TCP header encapsulated in any protocol, in turn encapsulated in PPP. Notably, the sequence of protocols specified in the filtering expression could start anywhere in the packet, therefore it could be preceded and followed by any protocol repeated an unspecified number of times.

Repetition operators describe conditions in which one or more protocols may occur a variable number of consecutive times in a certain position of the packet. In particular, "`+`" means one or more occurrences of the given protocol, "$*$" corresponds to zero or more, while "`?`" means zero or one. For example, the filter `ip in vlan* in ethernet` accepts the packets having IP encapsulated in zero or more consecutive VLAN headers, preceded by an Ethernet header.

More complex filters based on protocol encapsulations are available as well.

For instance, `tcp.sport==80 in (ip.src!=10.0.0.1)+ in ethernet` matches packets having the TCP protocol encapsulated in a sequence of one or more consecutive IP headers, in turn encapsulated in Ethernet; furthermore, the TCP source port must be equal to `80`, while the source address of each IP header must be different from `10.0.0.1`.

The *header indexing* construct selects a particular occurrence of a protocol header within the packet; for instance, `tcp in ip%2.src==10.0.0.1` matches all packets having TCP directly encapsulated in the second IP header of the packet, whose source address must be `10.0.0.1`. The *tunneling constraint* requires instead that a particular protocol is encapsulated in a tunnel (e.g., `ipv6 tunneled`); note that NetPFL does not indicate when a protocol is involved in a tunnel, which is left to the protocol database language instead.

Finally, multiple conditions on packets can be defined using the Boolean operators `and` and `or`. For instance, the filtering expression `(ip.src==1.0.0.1 tunneled) and (ip.dst==192.168.0.1 tunneled) and tcp.sport==2501 and tcp.dport==80` identifies a TCP section encapsulated in a tunnel.

3.2. Finite State Automata with Predicates (pFSA)

A pFSA [3] is an augmented FSA in which a transition may depend on the input symbols *and* on the value of a Boolean predicate associated with the transition itself; therefore it is called *transition with predicates* or *p-transition*. Notably, for each p-transition exiting from a state and firing if a predicate `p` is satisfied, there exists another p-transition exiting from the same state and that is triggered in case such a predicate is not satisfied.

When the pFSA is used to model filtering expressions, each state is associated with a network protocol and it is reached when that protocol is encountered inside the packet under analysis[1]. Instead, predicates associated with transitions

[1]Two exceptions hold for this rule: *(i)* the starting state, which is associated with a "dummy" protocol (called *Startproto* in the following) and representing the state of the automaton before starting the analysis; *(ii)* the state representing the non-accepting condition, not associated with any protocol and reached if the processed packet does not satisfy the filter.

consist of "basic blocks" in the form ‘‘`protocol.field operator value`’’, possibly combined together with Boolean operators (e.g., `tcp.sport==80 or tcp.dport==80`), which allow the transition to fire if the specified conditions are satisfied.

Pairs of p-transitions are modeled with another FSA, which sits on top of the base automaton and is evaluated when a predicate is encountered during the execution of the pFSA, in order to determine which of the two transitions must be triggered.

Figure 1 shows the pFSA representing the NetPFL filter `ip.src==10.0.0.1`[2], which is matched if *at least one* of the IP headers of the packet satisfies the condition, leading to the final accepting state (double-circled). When the base pFSA reaches a pair of p-transitions, the control is transferred to the pFSA sub-automaton implementing the associated predicate; the resulting value (true or false) is then returned to the main automaton in order to fire one and only one transition (the transition associated with the label `p` in case the predicate is satisfied, the one associated with the label `!p` otherwise).

The traditional algorithms and optimizations available for FSA have been extended for pFSA. Hence, if the filtering expression joins together multiple conditions through Boolean operators (e.g., `tcp.sport==58018 and tcp.dport==80`), a different pFSA is created for each condition, which are then combined in a single automaton representing the entire filter with optimality guarantees.

When a pFSA is used for packet filtering, the states of the automaton, its transitions and its set of input symbols derive from the protocol encapsulation rules defined in an external protocol database. Those rules can be represented with the *Protocol Encapsulation Graph (PEG)*, a directed, potentially cyclic graph that models the encapsulation relationships among protocols. As shown in Figure 2[3], each node of the PEG corresponds to a different protocol, while

[2]Note that, for the sake of clarity, the final non-accepting state and all the transitions that lead to it are omitted in Figure 1 and in any other automaton of the paper. Moreover, the symbol $*$ is used to indicate transitions that fire for every input symbol.

[3]Unless otherwise specified, the PEG in Figure 2 is referenced in all the examples of the

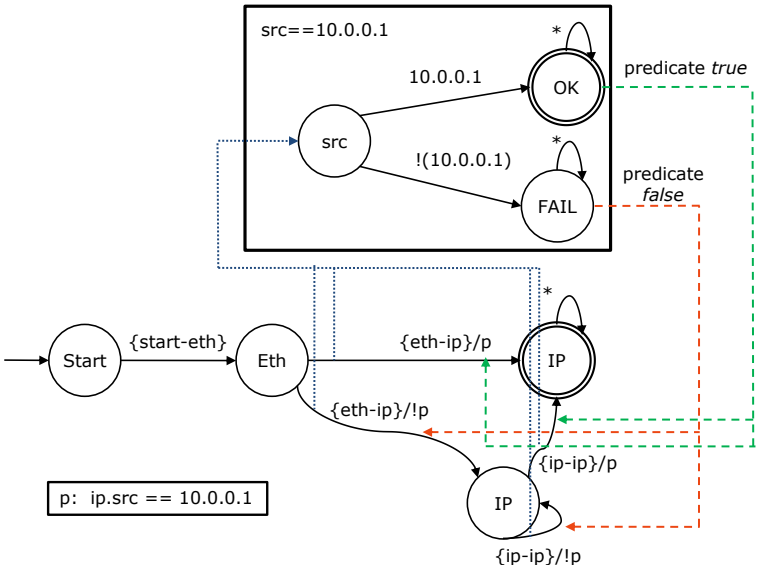

Figure 1: pFSA representing the NetPFL filter `ip.src==10.0.0.1`.

the edge from X to Y means that, within a packet, protocol Y could be directly encapsulated into X. Each symbol of the pFSA alphabet represents a different encapsulation rule and its name comes from the involved protocols: the name of the originating one first, the target last (e.g., the symbol `eth-ip` represents the encapsulation rule among Ethernet and IP).

Note that not all input symbols can be received while the control of the pFSA is in a given state. For instance, the symbol `eth-ip` can be only received when the pFSA is in the state associated with Ethernet, while the symbol `tcp-http` can be received only while the automaton is in the "TCP" state.

4. eXtended FSA with Predicates

An **eXtended Finite State Automaton with Predicates** (**xpFSA**) is an extension of the pFSA model that: *(i)* associates input symbols with operations on counters; *(ii)* associates p-transitions with predicates that check

paper.

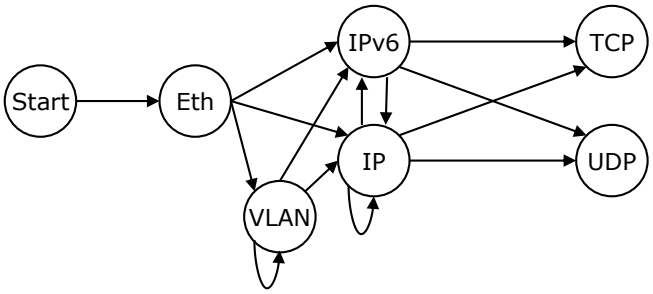

Figure 2: Example of Protocol Encapsulation Graph (PEG).

the value of such counters. This reduces the number of duplicate states and hence the cost of composition algorithms, which often depends on the number of states.

Formally, an xpFSA is defined with the following seven-tuple:

$$A_{xpfsa} = (Q, C,\ \Omega_c,\ \Sigma_o,\ \delta_p,\ q_0,\ F)$$

where:

Q is a finite set of *states*;

C is a finite set of *counters*;

$\mathbf{\Omega_c}$ is a finite set of *operations* on the counters defined in C;

$\mathbf{\Sigma_o}$ is the set of *input symbols*, each one potentially associated with a set of operations among those defined in $\mathbf{\Omega_c}$;

$\delta_\mathbf{p}$ is the *transition function with predicates* (p-transition), where possible conditions can take into account also the counters defined in C;

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

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

As in pFSA, each state Q of an xpFSA used for packet filtering is associated with one network protocol (with the exceptions already mentioned in

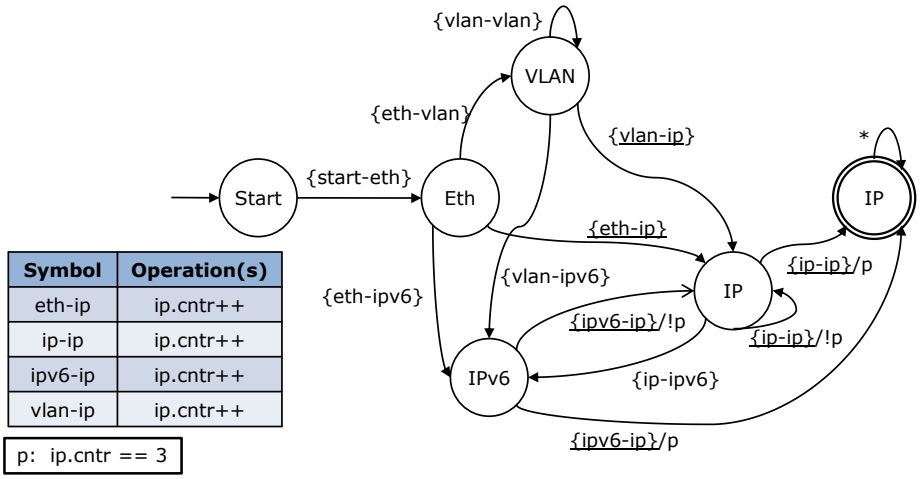

Figure 3: xpFSA modeling the NetPFL filter `ip%3`.

footnote 1); instead, input symbols originate from the encapsulations available in the PEG and can be potentially associated with a set of operations on counters, among those defined in Ω_c. When the control of the automaton receives a symbol, it performs the required operation(s) before triggering the proper transition(s).

An example of xpFSA using counters is depicted in Figure 3, which models the NetPFL rule `ip%3` and is built referring to the PEG of Figure 2. This automaton associates the operation `ip.cntr++` with the symbols having IP as a target protocol (i.e., all the symbols underlined in Figure 3); hence, each time that one of these symbols is received, the variable `ip.cntr` is incremented. Moreover, some p-transitions evaluate the value of this counter to determine whether the final accepting state can be reached or not. Notably, the same xpFSA shown in the picture can be used to evaluate all filters `ip%n` by simply changing the value of `ip.cntr` checked in predicate `p`.

4.1. Determinism and FSA-related algorithms

Being xpFSA derived from pFSA, most of the definitions and algorithms presented in [3] are valid here as well. For instance, an xpFSA is *deterministic* if it does not include any ϵ transition (i.e., transitions that do not require any

input symbol to fire) and, for each input symbol and for all possible values of the Boolean predicates, there is exactly one enabled outgoing transition, which is the same definition given in [3]. The same applies with the algorithms for *complementation*, *determinization* and *minimization*; instead, the *union* algorithm requires a new definition as it is influenced by counters introduced in xpFSA. Particularly, the union algorithm is extended as follows: the set of counters of the automaton resulting from the union of two xpFSA, is the union of the sets associated with the two contributing automata. Each of the input symbols of the resulting automaton is then associated with the union of the sets of operations associated with the same symbol in both the original xpFSA. Finally, the *intersection* can be implemented through the complementation and union algorithms according to the first De Morgan's law, hence it is only indirectly influenced by extensions defined in xpFSA.

5. Modeling filtering expressions through xpFSA

The generation of the filtering code implementing a given filtering expression is a complex process that requires the execution of several steps, which are summarized in left side of Figure 4. As shown, the filtering expression is first transformed into an automaton representing the filter itself (e.g., pFSA, xpFSA), by taking into account all the possible encapsulations that may be found in analyzed packets and that are described through a PEG. After creating the automaton, the corresponding code must be emitted, which implements the behavior of the automaton itself. Such a code must then be optimized, in order to finally obtain an efficient packet filter that can be exploited to analyze and filter packets flowing in a given portion of the network.

Particularly, this section focuses on the *automaton generation step*, which corresponds to the first step in Figure 4, and presents our algorithm that, starting from a protocol database modeled as a PEG and a filtering expressions *potentially* including constraints on protocol encapsulations[4], creates the corre-

[4]In fact, our algorithm can create automata modeling both traditional filters expressing

sponding xpFSA, which is later used to generate the executable code actually implementing the packet filter.

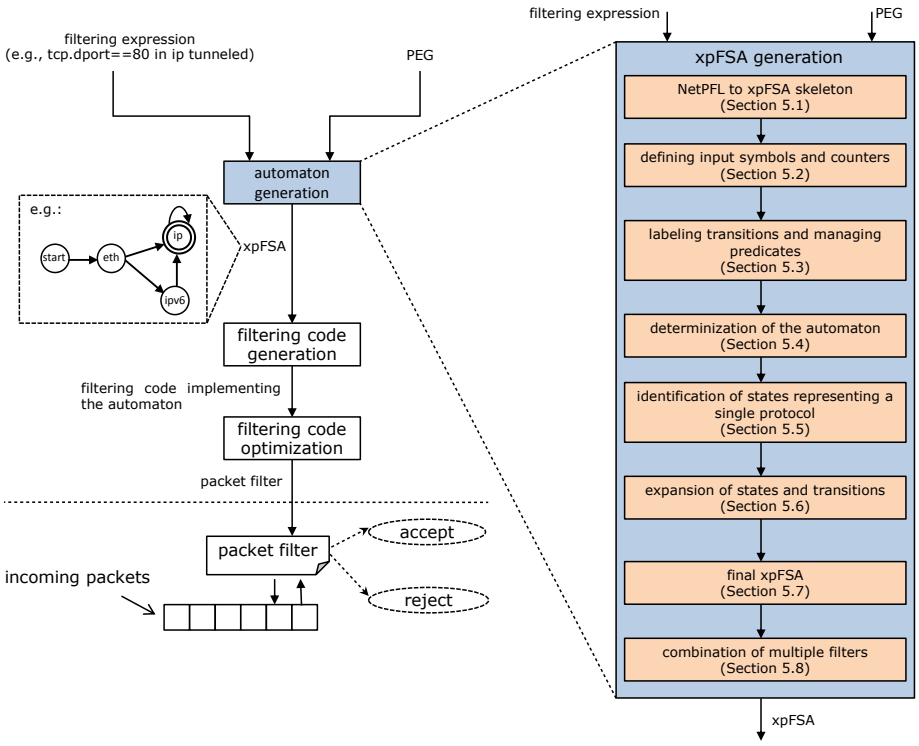

Figure 4: From the filtering expression to the packet filter: an overall view of the process.

The proposed algorithm deeply differs from the one defined in [3] (Section V-E) to build pFSA, which basically obtains the automaton by removing all the useless edges from the PEG, and potentially enriches transitions with sub-automata evaluating predicates expressed on protocol fields. As a matter of fact, although the pFSA formalism is able per se to model filters with protocol encapsulation constraints (even if, in some cases, not optimized in the number of states), the creation algorithm defined in [3] cannot create the automaton

constraints on protocol fields (e.g., `tcp`, `ip.src==10.0.0.1 and ip.dst==10.0.0.2`) and filters that specify protocol encapsulation rules (e.g., `tcp.sport==80 in ip`, `ip%2`).

actually representing such filters. As an example, the filter `tcp in ip in ip` requires at least two consecutive instances of IP before TCP, and this cannot be modeled by just *pruning* the undesired edges from the PEG.

As shown in right of Figure 4 and detailed in the remainder of this section, the new building algorithm is rather complex and has to follow a different approach to create the xpFSA: it first builds a basic automaton by considering the filtering constraints, which is then enriched according to the information described in the PEG.

5.1. NetPFL to xpFSA skeleton

The goal of this step of the algorithm is to create a first automaton out of the NetPFL filtering expression and then associate each state with one or more protocols. This automaton is created starting from the `in/notin` constructs, while the header indexing, the tunneling constraint and the predicates on protocol fields will be considered later.

To convert the filtering expression into an automaton, the NetPFL statement is split in a number of `tokens`, each one defined as:

```
[in|notin] {proto1,...} [repetitionOp]
```

where the elements have the same meaning introduced in Section 3.1.

Each `token`, starting from the rightmost one, is then converted in a different block of the automaton through the translation rules depicted in Figure 5, which come from the automata theory [12]. All the blocks are then connected in order, and the rightmost state represents the accepting state of the automaton. Since the header chain can be matched everywhere into the packet, a further state is added at the beginning of the automaton[5], which includes a self loop firing with any input symbol; an identical self loop is then added over the accepting state.

The new leftmost state is associated with all the protocols of the PEG, as each protocol is potentially allowed before those matching the header chain.

[5] We can optimize filters having Startproto in the rightmost position: since this protocol represents the beginning of the packet, this state is omitted.

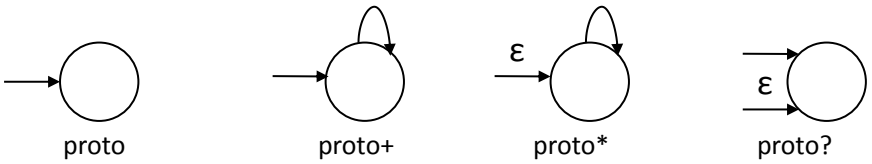

Figure 5: Building blocks of the automaton.

Each other state is instead associated with the protocols contained in the `token` from which it derives in case of the keyword `in`, or with all the protocols of the PEG excluding those listed explicitly in that `token` in case of the keyword `notin`. If the state is reachable also through an ϵ transition, the above protocols are integrated with those associated with the preceding state. In fact, whenever the automaton control is in the origin state of the ϵ transition, it is also in the target state of such a transition; therefore, its destination state is reached also when a protocol leading to its source state is encountered within the packet.

Figure 6 depicts the xpFSA skeleton built from the NetPFL filtering expression `tcp.sport==80 in {ip,ipv6}+ in ethernet` and from the PEG shown in Figure 2. Moreover, it shows the `token` from which each building block derives (at the top), and the protocols represented by each state (in the boxes at the bottom).

5.2. Defining input symbols and counters

Input symbols derive from the protocol encapsulation rules available in the PEG. For example, the edge (in the PEG) from IP to IPv6 originates the symbol `ip-ipv6`, which will be received by the automaton if the IPv6 header is directly encapsulated in IP within the analyzed packet. Each input symbol is then associated with a (potentially empty) set of operations to be executed when such a symbol is received, before triggering the proper transition(s).

Particularly, to model filtering expressions it is enough to *increment* counters. As described in the following, the rules for defining counters and for associating the operation(s) with the proper input symbols depend on whether the filtering expression has to recognize a specific header instance (defined through

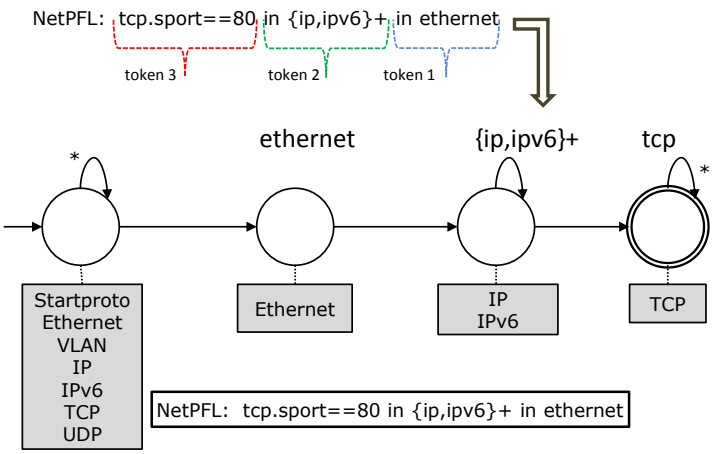

Figure 6: Skeleton of the xpFSA.

the header indexing construct) or it has to identify a protocol involved in a tunnel (specified through the tunneling constraint).

In fact, for each header indexing that refers to a different protocol, a new counter is created, whose name is in the form `proto.cntr`. The increment of this variable is then associated with those input symbols representing the identification of an instance of `proto` inside a packet. Considering the filter `ip%2`, the variable `ip.cntr` is defined and the operation `ip.cntr++` is associated with input symbols leading to IP, such as `eth-ip` and `ip-ip`.

Instead, each tunneling constraint that refers to a protocol of a different layer originates a counter whose name is in the form `Ln.cntr`, where n is a number representing the network layer of the protocol that must be involved in the tunnel. The operation `Ln.cntr++` is then associated with all symbols leading to a protocol belonging to a layer greater than, or equal to n[6]. As an example, consider the filter `ip tunneled`; IP belongs to layer 3, then the counter `L3.cntr` is defined and the operation `L3.cntr++` is associated with symbols leading to

[6]In fact, a protocol header is encapsulated in a tunnel if the layer of at least one of the protocol headers preceding it in a packet is greater than, or equal to, the layer of the protocol that is being considered.

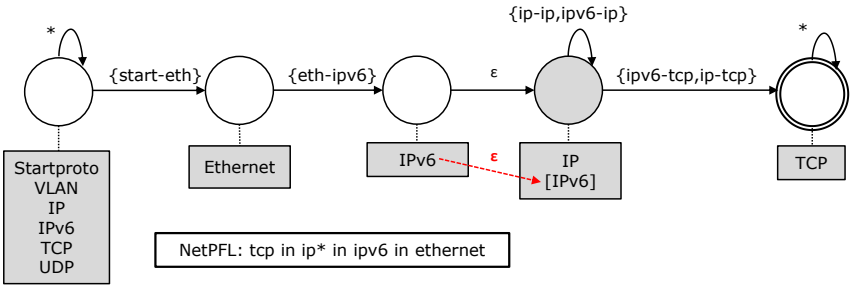

Figure 7: Transitions labeling process.

protocols corresponding, at least, to layer 3 (e.g., `eth-ip`, `ip-ipv6`). It is worth noting that assigning a layer to each protocol may not be trivial; more details will be presented in Section 6.

5.3. Labeling transitions and managing predicates

This step associates each *non-*ϵ transition with input symbols (defined according to a specific PEG) and, potentially, with predicates.

A transition is labeled with all the input symbols whose name satisfies the following constraints: *(i)* the origin protocol is equal to one of the protocols associated with the source state of the transition; *(ii)* the target protocol is equal to one of the protocols specified by the NetPFL `token` from which the destination state comes from. Hence, protocols associated with a state because of the ϵ transition cannot be target of the symbols leading to that state, as this could cause the recognition of wrong packets.

For example, the dark state in Figure 7 is associated with IP and IPv6 but, 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 recognizes only sequences of protocols matching the filter (e.g., Ethernet - IPv6 - TCP, Ethernet - IPv6 - IP - TCP, Ethernet - IPv6 - IP - IP - TCP). Instead, if IPv6 were the target of the symbols on the self loop, the automaton would also accept sequences such as Ethernet - IPv6 - IP - IPv6 - IP - TCP, which does not satisfy the filter to be modeled.

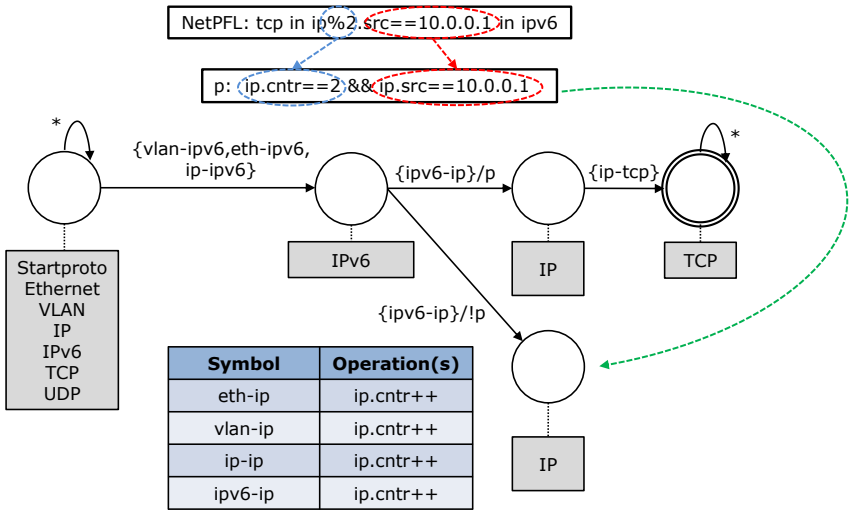

Figure 8: Managing predicates.

An exception to these labeling rules is represented by the self loop on the accepting state, which fires with any input symbol regardless of the protocol associated with the state itself.

Transitions that remain unlabeled (because no input symbol derived from the PEG satisfies the above rules) are removed from the automaton, since they can never fire.

Constraints on protocol fields, the header indexing construct and the tunneling constraint originate predicates to be evaluated on transitions. For example, the predicate `p` in Figure 8 derives from the requirements on the source IP address and on the header indexing expressed on such a protocol. As shown, the predicate is assigned to the transition leading to the state associated with the protocol involved in the predicate itself (IP in the example); moreover, a new state is created in the automaton, which is associated with the same protocol but that is reached through a transition firing if the predicate is not satisfied.

Another example is depicted in Figure 9, which shows how the automaton representing the filter `ip tunneled` requires a p-transition towards the rightmost state, which fires only if at least another protocol of layer greater than, or

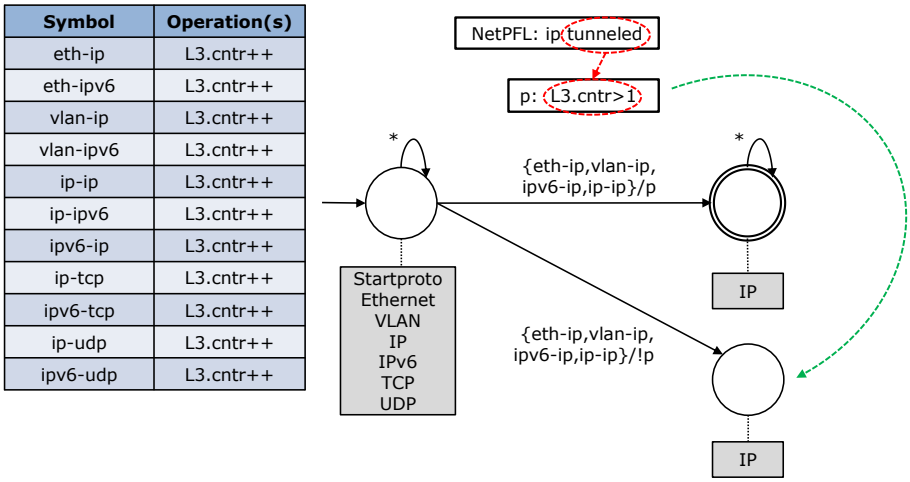

Figure 9: Deriving p-transitions form the tunneling constraint.

equal to 3 has already been encountered within the packet. Again, a new state associated with IP is added to the automaton and reached in case the predicate is not satisfied.

As described in [3], each predicate is actually modeled with a sub-automaton built on top of the xpFSA/pFSA; then, all the sub-automata associated with the same p-transition are combined using the traditional composition algorithms defined in the automata theory, which enable to obtain optimized xpFSA even in case of multiple filtering conditions expressed on the same protocol.

The process described above labels the transitions of the automaton of Figure 6 as depicted in Figure 10; as shown, a predicate is associated with transitions originating in the third state, because of the requirement on the source port of the TCP header.

5.4. Determinization of the automaton

The automaton created so far is now determinized according to the rules defined in [3]. As an example, the determinization process transforms the automaton of Figure 10 into the one depicted in Figure 11.

Unfortunately, not only states may be associated with multiple protocols, but

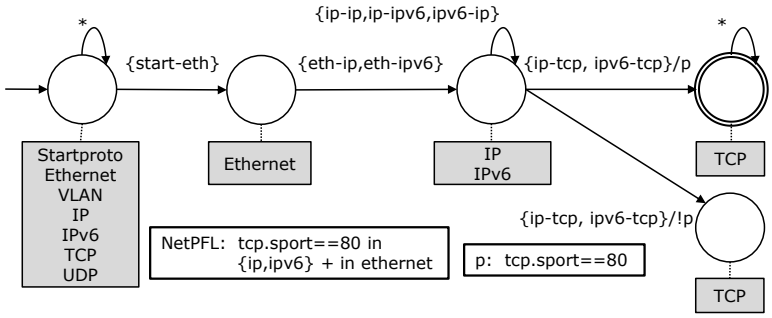

Figure 10: Automaton with labeled transitions.

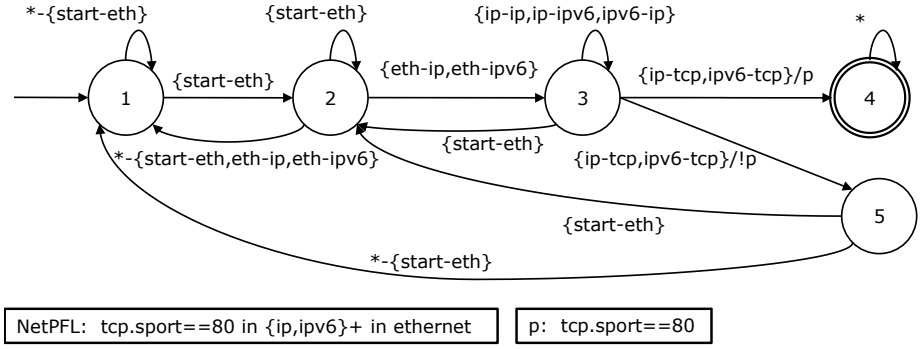

Figure 11: Deterministic automaton representing the filtering expression.

the state-protocol association might have been lost during the determinization process, as evident from Figure 11, which does not show any correspondence between protocols and states. Then, the next steps of the algorithm manipulate the automaton until each state is associated with one and only one protocol, so that reaching a certain state corresponds to reaching a specific protocol within a packet. This is needed to translate the xpFSA into actual executable code that can analyze and filter network traffic.

5.5. Identifying states representing a single protocol

A state corresponds to a specific protocol if all the symbols on its incoming transitions share the second part of their name, i.e. the target protocol of the encapsulation rules they represent is the same. Two exceptions are: *(i)* the initial state, which can be associated with `startproto` only if it does not have any incoming transition; *(ii)* the accepting state, whose self loop is (again) not considered. Each state that can be unequivocally associated with one and only one protocol is then labeled with the protocol it represents.

After that a state has been labeled, the symbols on its outgoing transitions are removed if their originating protocol differs from the one associated with the state itself. This is possible because symbols represent protocol encapsulation rules; hence, if the current state is associated with IP, only the symbols leading to a protocol encapsulated into IP can be received while the xpFSA is in that state. Obviously, transitions remaining without symbols, and states that cannot be reached from the starting state of the automaton or that do not lead to any accepting state, are removed.

This step transforms the automaton of Figure 11 into that shown in Figure 12, where states `2`, `4` and `5` are associated with a specific protocol.

5.6. Expanding states and transitions

Each unlabeled state U is now expanded in multiple states, each one associated with a different protocol among those that are the target of the symbols on the transitions leading to U. To model the situation in which the analysis

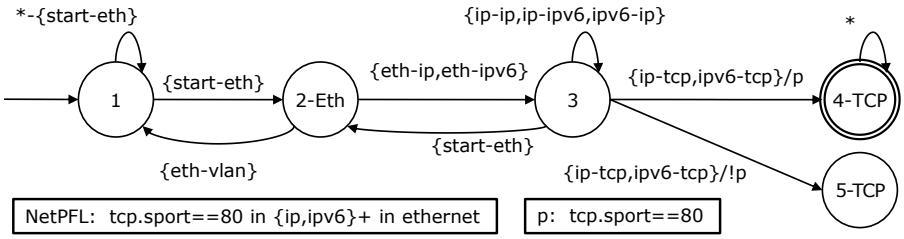

Figure 12: Automaton after the association of some states with a specific protocol.

of the packet has not started yet, the initial state also originates a new state associated with `startproto` and representing the new initial state of the automaton, although no input symbol actually leads to such a protocol. As an example, the dark state in the left of Figure 13 originates two new states in the right, respectively representing the protocols IP and IPv6.

Each transition *exiting* from an expanded state is replaced with a new transition for each one of its symbols. In particular, each new transition starts in the new state representing the origin 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 13 originates two new transitions: one labeled with `ip-ipv6` 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 protocol of their symbols. This way, the transition labeled with {`eth-ip`,`eth-ipv6`,`vlan-ip`,`vlan-ipv6`} originates two transitions: one firing with `eth-ip` and `vlan-ip` and leading to the new state representing IP, the other labeled with `eth-ipv6` and `vlan-ipv6` and entering into the new state associated with IPv6. Figure 13 also shows that the *self loop* on an expanded state originates new transitions that start and terminate on the proper new states, according to the origin and the target protocol of their symbols.

Figure 14 depicts the automaton of Figure 12 after this step of the algorithm, where states `1` and `3` have been expanded into multiple states. The symbols on the new transitions are not specified for the sake of brevity, and they can be

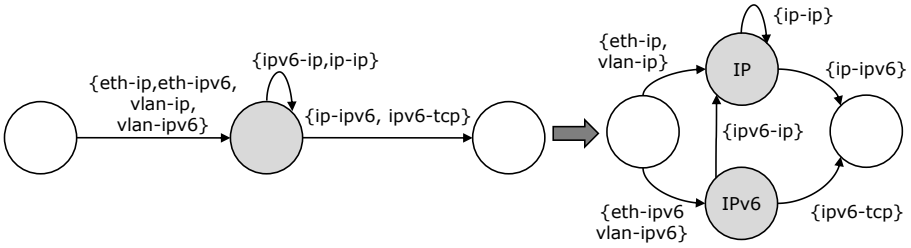

Figure 13: Expansion of a state and the related transitions.

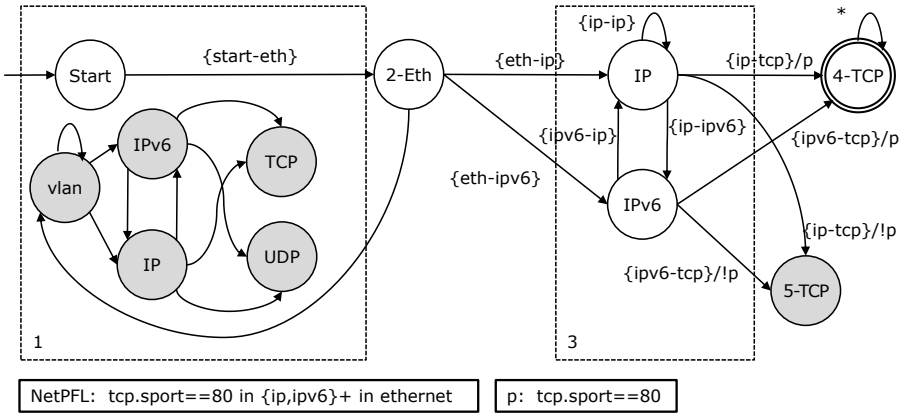

Figure 14: Automaton after the expansion of unlabeled states.

easily derived from the protocols labeling the states.

5.7. The final xpFSA representing the filtering expression

The xpFSA created so far may include some states that do not have a path to any accepting state: consequently, they can be removed without any loss in the semantic of the automaton. In some cases, the result of predicates operating on protocol counters may already be known a priory, i.e., before the generation of the filtering code. Then, transitions associated with predicates that are never verified are removed from the automaton, while predicates that are always verified are removed from the corresponding transitions in order to avoid useless checks.

After the pruning of the useless (dark) states and transitions, the automa-

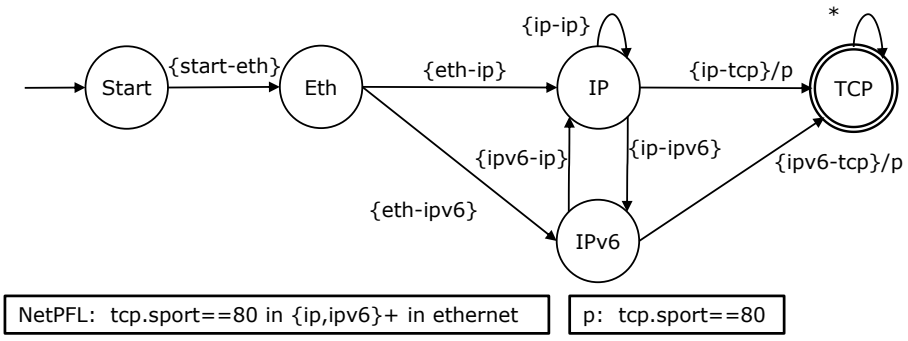

Figure 15: Final xpFSA representing the NetPFL filtering expression.

ton of Figure 14 becomes as shown in Figure 15, which is the final xpFSA representing the NetPFL filter `tcp.sport==80 in {ip,ipv6}+ in ethernet`.

5.8. Combining multiple filters

In case the NetPFL filter is composed of multiple conditions combined through Boolean operators, the algorithm presented so far is executed for each of them and the resulting automata are joined using the composition algorithm(s) discussed in Section 4.1.

An example is shown in Figure 16, which depicts the xpFSA modeling the filtering expression `(tcp.sport==80 in {ip,ipv6}+ in ethernet) or ip%2`[7]. Particularly, the automata representing the two parts of the filter are shown in Figure 16(a) and Figure 16(b), while Figure 16(c) shows the xpFSA modeling the entire expression. The set of counters in the final xpFSA is given by the union of the set of counters associated with the two contributing automata; then, it is associated only with `ip.cntr`, needed to count the number of IP header instances found in the packet. Moreover, each input symbol is associated with the union of the sets of operations associated with the same symbol in the original xpFSA; in fact, all the symbols with IP as a target protocol execute

[7] In order to get a more readable xpFSA, this automaton has been build referring to a PEG similar to that of Figure 2, but without VLAN.

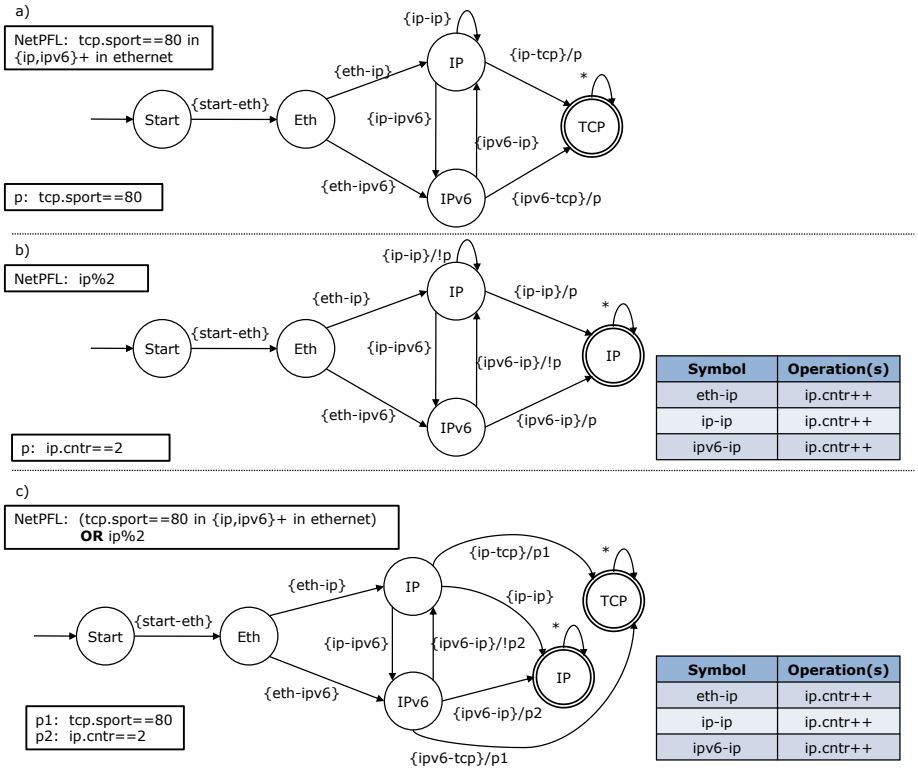

Figure 16: Composition of xpFSA with protocol encapsulation constraints through the Boolean `or` operation.

the operation `ip.cntr++` in the automaton of Figure 16(c).

6. Automatic detection of tunneled protocols

In case the filtering expression to be modeled with xpFSA includes tunneling constraints (e.g., `tcp in ip tunneled`, matching packets with the TCP header encapsulated in an IP header instance involved in a tunnel), the algorithm presented in Section 5 requires that each protocol in the PEG is associated with a layer. In fact, a protocol header is encapsulated in a tunnel if the layer of at least one of the protocol headers preceding it in a packet is greater than, or equal to, the layer of the protocol that is being considered.

At the first sight, the 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 for which it might be difficult to choose the "right" layer. A good example is MPLS, which may be present between Ethernet and IP; hence, 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 protocol is added to the database (e.g., VLAN, which may lead to the MPLS in VLAN encapsulation), as this operation may require an update of the values assigned to protocols already in the database.

Starting from the observation that, to recognize tunnels, the exact layer associated with each protocol is not important *per se* but only when compared to layers of other protocols, we propose an algorithm based on the PEG that provides a strict ordering based on network protocols. For example, to identify a tunneled IP, it is necessary that both IP and IPv6 are associated with the same layer, but it is not important the actual value of such a layer.

The algorithm acts as follows: *(i)* the layer value for all nodes in the graph is 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. Particularly, method `GetMinSuccessor` returns the smallest layer among those of a protocol successors, while `GetMaxPredecessor` returns the greatest layer among those of a protocol predecessors. In both cases, self loops are not considered. More in detail, according to lines 20-24 of Algorithm 1, each successor of the considered node is associated with a layer that is equal to the current layer plus one, in case this new value is lower than the layer already associated with that node. Then, if a node gets a layer that is equal to, or greater than the smallest layer among those of its successors, the layer value for the node is potentially updated according to lines 11-17. The algorithm terminates when all the nodes of the PEG are associated with a layer, and the outcome does not depend on the order in which the nodes of the PEG

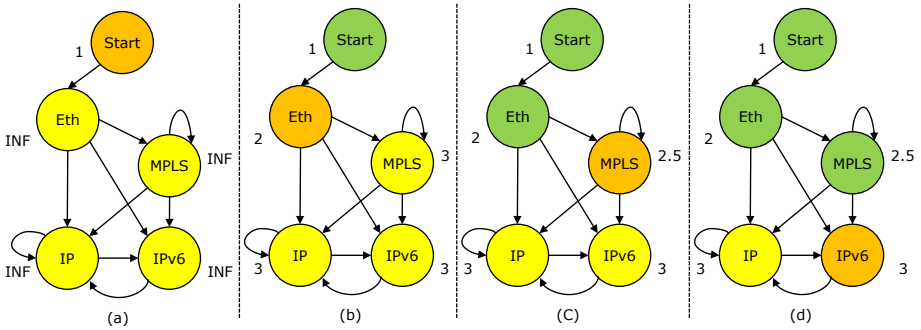

Figure 17: Layer assignment example.

have been visited.

The example in Figure 17(a) depicts a PEG where all protocols (except Startproto) are not associated with any layer. First, the procedure assigns to the successors of Startproto, in 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, as shown in Figure 17(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 as shown in Figure 17(c). Because of the check in line 21 of Algorithm 1, the successors of MPLS are not updated. Finally, IP and IPv6 are considered but, since `prevlevel = nextlevel = 3` for both of them, line 15 of the algorithm is not executed and their layers remain unchanged (Figure 17(d)).

7. Implementation

The proposed algorithm that transforms filtering expressions with protocol encapsulation constraints into xpFSA has been integrated in the NetBee library [4], whose overall architecture is shown in Figure 18. User-level tools (e.g., `nbeedump`) receive as input the NetPFL rule to be executed and the NetPDL [5] protocol database. This information is then processed by an high-level compiler [21] that, after several optimizations, emits the final filtering code un-

Algorithm 1 Classifying each protocol of the PEG.

```
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 ∈ n.successors do
27:    AssignProtoLevels(s)
28: end for
```

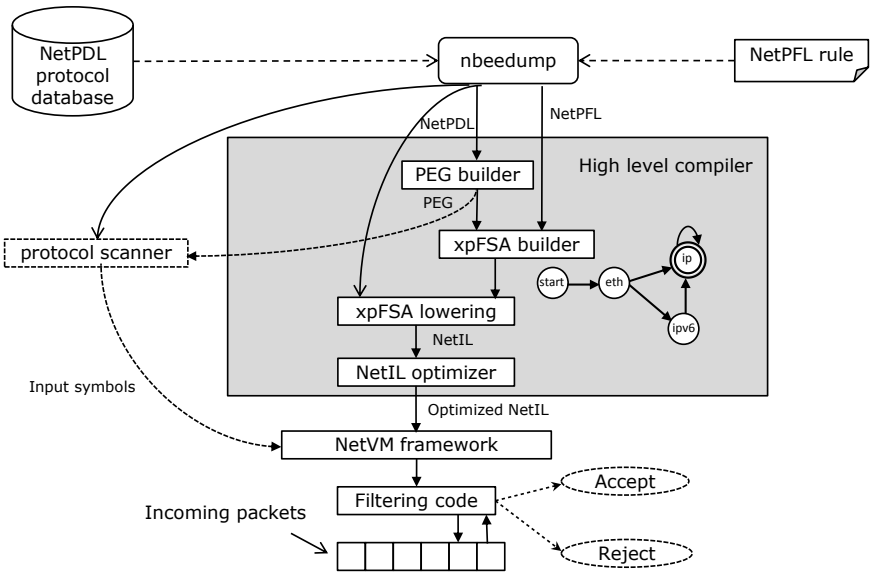

Figure 18: Overview of the building blocks to generate filtering code.

der the form of NetIL instructions, i.e., the assembly for the NetVM [22] virtual machine. NetIL can be either interpreted by the NetVM or translated in native code for different architectures (e.g., x86, x64, Cavium Octeon) thanks to a multi-target compiler [23].

The algorithm presented in Section 5 is implemented in the *xpFSA builder* module, which takes both the PEG (dynamically extracted from the NetPDL database by the *PEG builder* module, which also associates each protocol with a layer using the algorithm described in Section 6) and the NetPFL rule and builds the corresponding xpFSA. Then, the *xpFSA lowering* module generates the corresponding NetIL code by translating each state of the automaton according to the NetPDL description of the protocol associated with the state itself. Although the input symbols for the automaton are generated by a logical separated module (i.e., the *protocol scanner*), its operations are implemented by the same assembly program implementing the xpFSA, hence the NetIL code corresponding to an xpFSA state contains both the code that implements the automaton and the one that handles the encapsulations. Finally, the *NetIL*

optimizer executes a set of data and control flow optimizations on such a representation; the resulting code is given as an input to the NetVM and, possibly, further translated into native code (e.g., x64).

8. Validation

The work of this paper has been evaluated through a number of tests, described in the remainder of this section.

8.1. The influence of the protocol database

Before going into detail of our evaluation, it is worth to point out that the time needed to create the automaton, and the total time required to generate efficient filtering code out of xpFSA depend on the complexity of the protocol database, and hence on the PEG of reference. In fact, the former time increases with the size of the PEG, which influences both the number of states and the number of transitions exiting from each state. In turn, more states cause the generation of more executable code while, at runtime, the number of outgoing transitions may influence the time needed to determine which is the next protocol of the packet. The code generation time also depends on the format of each protocol, as it grows with the number and the complexity of protocol fields (e.g., variable length fields require the generation of more instructions than fixed length fields).

In our evaluation campaign we use several PEGs, which will be detailed in each specific test. However, in all the cases we use the official protocols description provided with the NetBee library [4].

8.2. xpFSA creation, filtering code generation and filtering throughput

This section evaluates the performance of our algorithm to generate the xpFSA that model filtering expressions with constraints on protocol encapsulations (Section 5), and the total time required by the NetBee library to generate

efficient filtering code from xpFSA. Moreover, since our final goal is the precise filtering of packets, we also report the runtime performance of such packet filtering code.

Notably, at the best of our knowledge no packet filter supports filtering expressions with constraints on protocol encapsulations, then we do not compare the results of this section with those obtained through other approaches.

The sample filters are shown in Table 1, while Figure 19 reports the PEG of reference and shows the network layer associated with each protocol, calculated through the algorithm presented in Section 6. This PEG is quite realistic, since it includes several encapsulations and protocols commonly encountered nowadays on the Internet. Finally, tests are executed on a workstation equipped with 16 GB RAM, 1TB hard disk @ 7200 rpm, Intel i7-3770 @ 3.40 GHz CPU and Ubuntu 14.04 OS, kernel 3.13.0-49-generic, 64 bits. All test processes were bound to a single processor, with hot disk and processor caches, and the machine was otherwise unloaded. Time measurements were performed using the `gettimeofday` UNIX function.

8.2.1. Filtering code generation time

According to Figure 20, the time required by our algorithm to create the xpFSA is at least one order of magnitude lower than the time needed (by the NetBee library) to generate the optimized NetIL code implementing the specific NetPFL rule, for almost all the considered filters.

Moreover, the picture highlights how the code generation time decreases by increasing the selectivity of the filter, while the time needed to create the xpFSA follows the opposite trend. For instance, this can be observed by comparing the sequence of filters `#1`, `#2` and `#3`, or the sequence `#1`, `#5` and `#6`, which both define filters with an increasing degree of selectivity for matching TCP packets. Notably, the generation of the automaton is slower if there are less protocols that could match the initial state in the xpFSA skeleton (Section 5.1). In fact, our algorithm first expands this initial state in most of the protocols defined in the PEG, then it prunes those states that are not necessary. Hence, less protocols

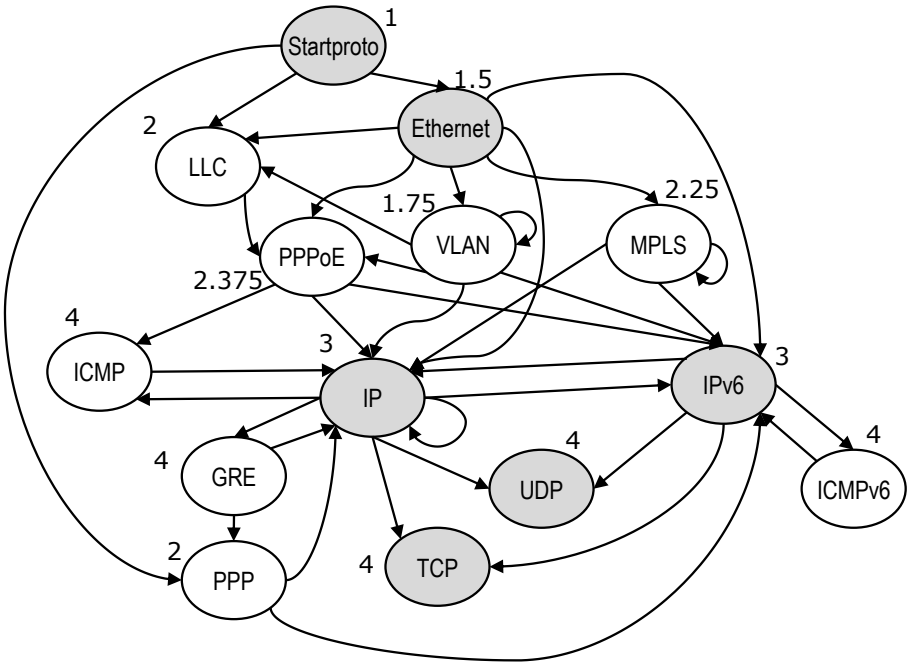

Figure 19: Protocol Encapsulation Graph.

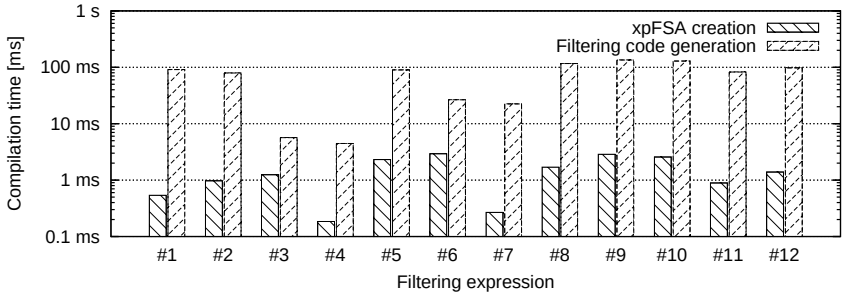

Figure 20: Performance of the code generator.

Table 1: Sample NetPFL filtering expressions with protocol encapsulation constraints, and percentage of packets matched in the traffic trace used in the evaluation.

#	Filtering expression	% of acceptance	#accepted packets	#expected packets
1	tcp	100	14045400	14045400
2	tcp in ip	66	9363600	9363600
3	tcp in ip in ethernet	33	4681800	4681800
4	tcp in ip in ethernet in startproto	33	4681800	4681800
5	tcp in ip in ppp in gre in ip	33	4681800	4681800
6	tcp in ip in ppp in gre in ip in ethernet	33	4681800	4681800
7	tcp in ip in ppp in gre in ip in ethernet in startproto	33	4681800	4681800
8	tcp in ip notin ethernet	33	4681800	4681800
9	tcp in ip tunneled	33	4681800	4681800
10	tcp in ip%2	33	4681800	4681800
11	tcp in ipv6	33	4681800	4681800
12	tcp in ipv6 in ip	33	4681800	4681800

matching this state mean more cuts in the automaton, which results in more time to create the xpFSA. Instead, filters that explicitly mention `startproto` (`#4` and `#7`) immediately generate very compact automata (they do not have the initial state to be expanded, since it is only associated with Startproto), and represent the fastest case for our algorithm that builds the xpFSA representing a filtering expression.

Scalability - Counters reduce the number of states of the xpFSA, hence the time required to generate the final code implementing the NetPFL filter. Figure 21 shows this reduction through filters that require an increasing number of IP headers within valid packets, by reporting the number of states and the filtering code generation time both in case the associated automata use counters and in case counters are not used. To this purpose, we modified our *xpFSA builder* module so that it can also implement the NetPFL *header indexing* con-

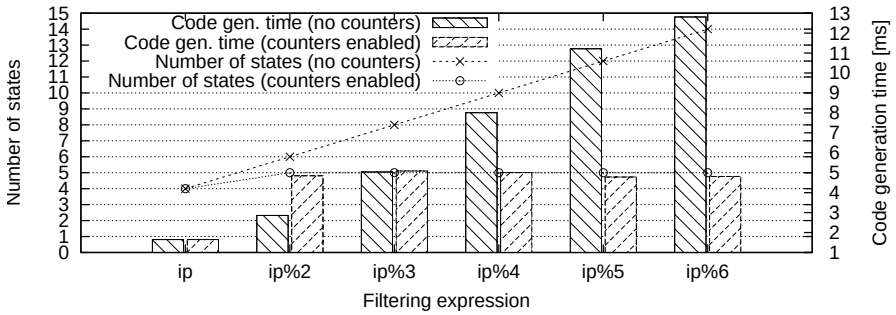

Figure 21: Advantage of using counters in xpFSA.

struct without using counters. The automata have been built using a PEG that includes only the dark states shown Figure 19.

Without counters, the number of states grows steadily with the number of required IP headers, since the automaton representing `ip%n` consists in the automaton associated with `ip%(n-1)`, enriched with all paths leading from IP to IP. Instead, with counters, 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. Then, in terms of complexity of the automaton, counters bring advantages for filters requiring that a specific protocol header appears at least twice in the packet.

Regarding the total compilation time, counters are beneficial from filter `ip%4` forward, and such an advantage increases with the number of required IP headers. Instead, in case of `ip%2`, the cost of managing counters exceeds that required to compile a further state in the automaton. With filter `ip%3`, the cost of counters is equivalent to that of managing three additional states, resulting in the same compilation time regardless of the fact that counters are used or not.

As a final remark, even if the reduction in the number of states, and hence of the compilation time, is minimal in our example, it could be substantial both in case of more complex filters, and in other fields different from packet filtering. In [13], the advantage of the reduction of the number of states through the assignation of instructions to the automaton has been demonstrated using the

XFA model in the field of string matching.

8.2.2. Filtering throughput

In order to evaluate the quality of the filtering code generated by xpFSA, namely the x64 assembly program that actually analyzes and filters packets, we executed the filters of Table 1 on a synthetic traffic trace composed of three packets of 700B[8] repeated as many times as needed to obtain about 9.15 GB of traffic. Those packets aim at reproducing some common encapsulations, namely `ethernet-ip-tcp`, `ethernet-pppoe-ppp-ip-ipv6-tcp` and `ethernet-ip-gre-ppp-ip-tcp`. Particularly, the second packet can be observed in an IPv4-only xDSL-based access network connecting to the Internet a client using IPv6 as a network protocol, while the last packet is used in a PPTP-based VPN.

The percentage of packets accepted by each filter is reported in the third column of Table 1, while the last two columns respectively show the number of packets accepted by the filter and the number of packets expected to be accepted. Since these two numbers always coincide, the filtering code generated from xpFSA is correct for each of the considered filters, and actually implements the constraints expressed in the filtering expression. Consequently, being the filtering code generated from xpFSA correct, it is also correct our algorithm presented in Section 5, which transforms the filtering expression into an xpFSA.

Figure 22 shows the number of packets per second analyzed by the sample NetPFL filters. Note that one-time computations (e.g., filter compilation) do not affect the result of this test, while runtime overheads (e.g., per-packet `libpcap` library call) are included in the results. As shown, performance increase when the filter is more specific, i.e., when it leaves less freedom to the protocols that may appear in a given position of the packet. Notably, according to the graph, encapsulation-aware filtering code can be applied on a 10 Gbps link without any packet loss, if considering an average packet size of 700 bytes.

[8] This value roughly represents the average packet size on the Internet.

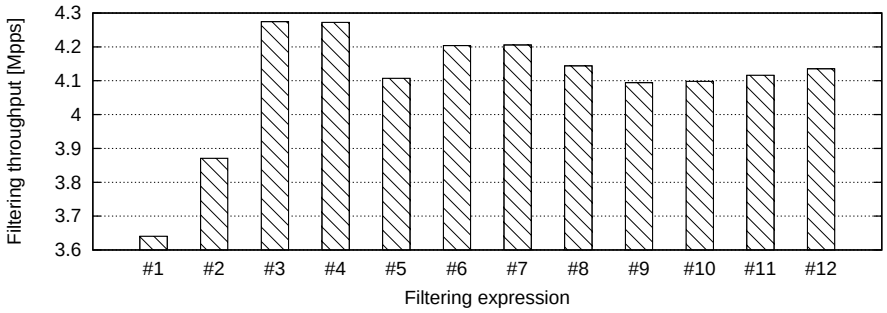

Figure 22: Performance of the filtering code generated by xpFSA.

Table 2: Filtering throughput with respect to TCP session filters.

# TCP sessions	**Throughput (Mpps)**
1	4.3
2	4.2
3	4.2

Scalability - In order to provide an insight of the scalability of the filtering code generated out xpFSA, we also executed a number of filters that select an increasing number of TCP sessions encapsulated in a tunnel. A single TCP session is represented by a filtering expression in the form `((ip.src==x.x.x.x tunneled) and (ip.dst==y.y.y.y tunneled) and tcp.sport==X and tcp.dport==Y)`, while the Boolean `or` operator is used to combine together filters matching different sessions.

Filtering code have been executed on the synthetic traffic trace mentioned before (each filter matched one third of packets), and results are reported in Table 2. As shown, filtering code generated by xpFSA does not suffer any significant runtime performance degradation when the number of filtered sessions increases. This is due to the fact that the generated automaton grows wider, but not deeper; as the number of sessions increases, more and more states are added in parallel to the old ones, but the average distance from the starting state to the accepting states does not change.

8.3. Support for traditional filters

The algorithm presented in Section 5 also creates xpFSA representing traditional filters based on the existence of some protocols and on the value of some protocol fields. Then, this section evaluates our algorithm when applied to those filters, and compares it with the algorithm proposed in [3] (Section V-E), which models (only) traditional filters with pFSA. It is worth mentioning that we compare our proposal with [3] because both of them: *(i)* represent filtering expressions through augmented FSA; *(ii)* exploit an external protocol database represented as a PEG. The comparison with other solutions for packet filtering (e.g., SPAF, BPF) is not repeated in this paper, as [3] already compares them with the algorithm defined for pFSA.

We repeated the same tests reported in [3] with the new algorithm; the sample filters are shown in Table 3, using two different PEGs. The *simple* PEG includes only the definitions for `Ethernet, IPv4, TCP` and `UDP`, without recursive encapsulations. The *complex* PEG includes also the definitions for `VLAN, ARP, PPPoE` and `IPv6`, together with the following tunnels: `IPv4-in-IPv4`, `IPv6-in-IPv4`, `IPv6-in-IPv4`. Finally, as in [3] tests have been executed on a workstation equipped with an Intel E8400 Core 2 Duo dual-core processor with 4 GB of RAM, running a 64-bit version of Ubuntu 10.04.

Table 3: Sample traditional NetPFL filters.

#	**Filtering expression**
1	ip
2	ip.src == 10.1.1.1
3	tcp
4	ip.src == 10.1.1.1 and ip.dst == 10.2.2.2 and tcp.sport == 20 and tcp.dport == 30
5	ip.src == 10.1.1.1 or ip.dst == 10.2.2.2 or tcp.sport == 20 or tcp.dport == 30

Figure 23 depicts, in logarithmic scale, the time needed by the algorithm proposed in [3] to create the pFSA, and by the new algorithm to create the xpFSA representing the filters of Table 3, both with *simple* and the *complex*

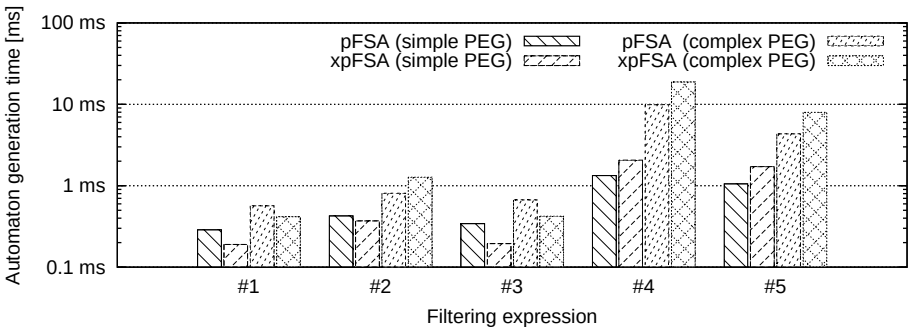

Figure 23: Comparison of the time needed by our algorithm and by that proposed in [3] to create the automaton out of a filtering expression.

databases. According to the picture, although the new algorithm is considerably more complex then its ancestor, it does not introduce considerable overhead when used to model traditional filtering expressions, and it even performs better in some cases.

Notably, since the filters in Table 3 do not include constraints on protocol encapsulations, the pFSA/xpFSA generated by the compared algorithm are identical. Consequently, they originate the same filtering code, which obviously provides the same performance (in terms of packets per second) when used to analyze network traffic.

9. Conclusion

Currently, no existing packet filter enables the precise filtering of network packets based on constraints on protocol encapsulation relationships; this prevents applications to finely select traffic they are interested in, e.g., in case of complex protocol encapsulations and/or tunnels. To address this limitation, this paper presents an algorithm that efficiently implements packets filters that include protocol encapsulation constraints.

The proposed algorithm supports filtering expressions that filter packets that may include: *(i)* specific header sequences (e.g., `tcp.sport==80 in ipv6`, `ipv6 notin vlan`); *(ii)* a specific value for a field in a given header instance (e.g.,

`ip%2.src==10.0.0.1`); *(iii)* a given protocol encapsulated in a tunnel (e.g., `ip.src==10.0.0.1 tunneled`, `ip tunneled`).

Particularly, such an algorithm, starting from a filtering expression written according to the Network Packet Filtering Language (NetPFL), and from a protocol database modeled as a Protocol Encapsulation Graph, models the filter by means of xpFSA, a novel formalism that extends the Finite State Automata with predicates (pFSA). xpFSA guarantees optimal composition of multiple filtering expressions and efficient packet filtering code. Moreover, it defines operations (i.e., the increment of counters) to be executed when specific input symbols are received; this causes a reduction of the number of states of the automaton and then a reduction of its complexity.

Finally, the paper proposes an algorithm that can be used to automatically recognize packets that include tunneled protocols, which is based on the dynamic association of each protocol to its supposed network layer.

Evaluation shows the efficiency of the algorithm that transforms filtering expressions into automata, the time required to generate filtering code out of xpFSA, and the runtime performance of the generated packet filtering code. We also show how our algorithm does not penalize traditional filters only based on constraints on protocol fields, as well as we evaluated scalability of the automaton and of the generated packet filtering code.

Acknowledgment

The authors would like to thank Marco Leogrande for the inspiring discussions and comments, and Olivier Morandi for the help in defining the algorithm to automatically assign protocols to layers.

References

[1] L. Ciminiera, M. Leogrande, J. Liu, F. Risso, O. Morandi, A tunnel-aware language for network packet filtering, in: Proceedings of the Global Com-

munications Conference (Globecom 2010), Miami, Florida, USA, 2010, pp. 1–6.

[2] 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, USA, 2013, pp. 1058–1063.

[3] M. Leogrande, F. Risso, L. Ciminiera, Modeling complex packet filters with finite state automata, IEEE/ACM Transactions on Networking 23 (1) (2015) 42–55.

[4] F. Risso, et al., The netbee library, version 0.4 (Apr 2016).
URL `https://github.com/netgroup-polito/NetBee`

[5] F. Risso, M. Baldi, Netpdl: an extensible xml-based language for packet header description, Computer Networks 50 (5) (2006) 688–706.

[6] S. McCanne, V. Jacobson, The bsd packet filter: a new architecture for user-level packet capture, in: Proceedings of the USENIX Winter 1993 Conference, 1993, pp. 2–2.

[7] G. Combs, et al., Wireshark display filters (Oct 2008).
URL `http://wiki.wireshark.org/DisplayFilters`

[8] J. Mogul, R. Rashid, M. Accetta, The packer filter: An efficient mechanism for user-level network code, in: Proceedings of the Eleventh ACM Symposium on Operating Systems Principles, SOSP '87, 1987, pp. 39–51.

[9] A. Begel, S. McCanne, S. L. Graham, Bpf+: Exploiting global data-flow optimization in a generalized packet filter architecture, in: Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, SIGCOMM '99, 1999, pp. 123–134.

[10] M. L. Bailey, B. Gopal, M. A. Pagels, L. L. Peterson, P. Sarkar, Pathfinder: A pattern-based packet classifier, in: Proceedings of the First USENIX

Symposium in Operating System Design and Implementation, Monterey, CA, 1994, pp. 115–123.

[11] D. R. Engler, M. F. Kaashoek, Dpf: Fast, flexible message demultiplexing using dynamic code generation, SIGCOMM Comput. Commun. Rev. 26 (4) (1996) 53–59.

[12] J. E. Hopcroft, R. Motwani, J. D. Ullman, Introduction to automata theory, languages, and computation (2. ed), Addison-Wesley, 2003.

[13] R. Smith, C. Estan, S. Jha, Xfa: Faster signature matching with extended automata, in: Proceedings of the 2008 IEEE Symposium on Security and Privacy, 2008, pp. 187–201.

[14] R. Sekar, P. Uppuluri, Synthesizing fast intrusion prevention/detection systems from high-level specifications, in: Proceedings of the 8th conference on USENIX Security Symposium - Volume 8, 1999, pp. 6–6.

[15] G. Van Noord, D. Gerdemann, Finite state transducers with predicates and identities, Grammars 4 (3) (2001) 263–286.

[16] 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, 2007, pp. 1–10.

[17] V. Laurikari, Nfas with tagged transitions, their conversion to deterministic automata and application to regular expressions, in: String Processing and Information Retrieval, 2000. SPIRE 2000. Proceedings. Seventh International Symposium on, 2000, pp. 181–187.

[18] C. Meiners, E. Norige, A. X. Liu, E. Torng, Flowsifter: A counting automata approach to layer 7 field extraction for deep flow inspection, in: INFOCOM, 2012 Proceedings IEEE, 2012, pp. 1746–1754. doi:10.1109/INFCOM.2012.6195547.

[19] H. Li, C. Hu, J. Hong, X. Chen, Y. Jiang, Parsing application layer protocol with commodity hardware for sdn, in: Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for Networking and Communications Systems, ANCS '15, IEEE Computer Society, Washington, DC, USA, 2015, pp. 51–61.
URL http://dl.acm.org/citation.cfm?id=2772722.2772732

[20] P. Rolando, R. Sisto, F. Risso, Spaf: stateless fsa-based packet filters, IEEE/ACM Transactions on Networking 19 (1) (2011) 14–27.

[21] 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, 2008, pp. 5849–5856.

[22] L. Degioanni, M. Baldi, D. Buffa, F. Risso, F. Stirano, G. Varenni, Network virtual machine (netvm): a new architecture for efficient and portable packet processing applications, in: Proceedings of the 8th International Conference on Telecommunications (ConTEL 2005), Zagreb (Croatia), 2005, pp. 163–168.

[23] O. Morandi, F. Risso, P. Rolando, S. Valenti, P. Veglia, Creating portable and efficient packet processing applications, Design Automation for Embedded Systems 15 (1) (2011) 51–85.