Filtering Network Traffic Based on Protocol Encapsulation Rules

Ivano Cerrato
Politecnico di Torino
Corso Duca degli Abruzzi 24,
Torino, Italy
Email: ivano.cerrato@polito.it

Marco Leogrande
Politecnico di Torino
Corso Duca degli Abruzzi 24,
Torino, Italy
Email: marco.leogrande@polito.it

Fulvio Risso
Politecnico di Torino
Corso Duca degli Abruzzi 24,
Torino, Italy
Email: fulvio.risso@polito.it

***Abstract*—Packet filtering is a technology at the foundation of many traffic analysis tasks. While languages and tools for packet filtering have been available for many years, none of them supports filters operating on the *encapsulation relationships* found in each packet. This represents a problem as the number of possible encapsulations used to transport traffic is steadily increasing and we cannot define exactly which packets have to be captured.**

This paper presents our early work on an algorithm that models protocol filtering patterns (including encapsulation constraints) as Finite State Automata and supports the composition of multiple expressions within the same filter. The resulting, optimized filter is then translated into executable code. The above filtering algorithms are available in the NetBee open source library, which provides some basic tools for handling network packets (e.g., a `tcpdump`-like program) and APIs to build more advanced tools.

I. Introduction

In the recent years we have observed a reduction in the number of layer-7 protocols in use. In fact, while in the past each application defined its own protocol, nowadays most of the traffic is conveyed through the web. As a consequence, 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 transforming the lower layers of the protocol stack into a mess. Figure 1 presents one of the possible examples of the complexity growing over the years, which translates e.g. 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.

Particularly, when operating at the upper layers (e.g., filtering based on TCP ports) it is important to be able to capture all the traffic we are interested in, independently from the actual encapsulations used at lower layers, be it plain Ethernet, VLAN in WiFi, MPLS, IPv6 in IPv4, GRE-tunneled, or anything else. While this looks simple in principle (essentially, we need to support more encapsulations when generating the actual filtering executable), it may not be easy to modify a filtering tool to handle more complex protocol encapsulations.

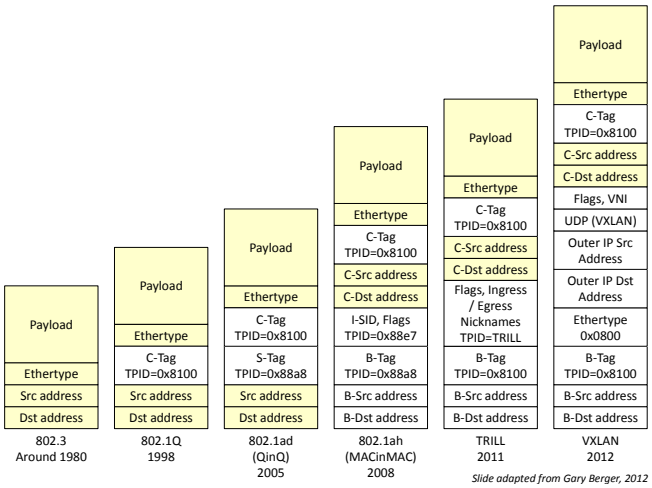

Fig. 1. Growing complexity in protocol encapsulations.

The NetPDL language [1] aims at solving this problem, by enabling the creation of tools in which protocol formats and encapsulations are no longer hardwired into the filtering program, but are in a *separate* XML file. Additional encapsulations can be added by simply editing the XML file, without any modification to the tool itself. This solution allows users to define high-level packet filtering rules (e.g., `tcp.sport == 80`), while the NetPDL-based tool will take care of selecting the desired traffic, whatever is the encapsulation that is being used. However, the capability to follow any possible encapsulation may result into slower processing (as the filtering code is forced to check for any possible encapsulation path) and this additional cost may not always be acceptable. A possible solution to this issue is provided by the NetPFL language [2], which allows to define packet filtering rules including which encapsulation paths have to be followed (e.g., `ip in vlan in ethernet`), without modifying the NetPDL protocol definitions.

While the NetPFL language is very flexible, so far only a partial implementation is available [2], which does not support the explicit filtering on protocol chains; consequently the possible optimizations were not taken into consideration at all. This paper extends the initial work by presenting an

algorithm that can generate efficient filtering code based on NetPFL *header chains*, that can select traffic based on one or more encapsulation rules specified at run-time. The NetPFL filtering expression is transformed into a formalism based on Finite State Automata (FSA), where states and transitions are derived by the NetPDL database in use. The FSA algebra offers the possibility to compose multiple filters, with a result that can be translated into a deterministic FSA (DFA) that guarantees the fastest matching path for that filter. Finally, that DFA is translated into the executable code that actually analyzes the network packets.

This paper is structured as follows. Section II presents the main concepts of the NetPFL language. The DFA building algorithm is presented in Section III, while an overview of the implementations is given in Section IV. A preliminary evaluation of the algorithm is shown in Section V, while Section VI concludes the paper.

II. The NetPFL language

The **Network Packet Filtering Language** (NetPFL) [2] is a declarative high-level language that can be used to define packet filtering rules. The NetPFL syntax does not define the list of protocols and fields supported; instead, they are dynamically bound to those defined in an external data set that, in our implementation, is based on the NetPDL language.

NetPFL is more complex than other existing packet filtering languages, as it allows to specify not only the *conditions* that a packet must satisfy in order to be accepted, but also the *actions* to be executed when a packet is accepted and the *stream* the packet belongs to, in order to support multiple filters at the same time. The filtering syntax is very similar to the one used by classical packet filters and it supports multiple conditions to be joined with the `and` and `or` logical operators. Moreover, a condition can be negated through the `not` operator.

In addition to the constructs mentioned above, which are fairly common across different filtering languages, NetPFL supports other primitives that operate on protocol encapsulations. Among those, there is the **header chains** feature, which is the focus of this paper.

A header chain defines a filtering condition based on protocol encapsulation rules that have to be satisfied when capturing the traffic. Its core elements are the keywords `in` and `notin`, which require respectively that, within a packet: *(i)* the left-hand element is directly encapsulated into the right-hand one, and that *(ii)* the left-hand element is encapsulated in any protocol other than the right-hand one. For instance, `tcp in ip`[1] accepts a packet defined as WiFi-IP-TCP, while rejects WiFi-IP-IPv6-TCP; `ip notin vlan` matches a packet such as Ethernet-IP, while discards Ethernet-VLAN-IP. An element of the header chain could be a *header set*, which specifies a *set* of protocols that can be (or must not be, in case of the `notin` keyword) in a given position of the encapsulation stack. For instance, previous examples make use of a single protocol, which can be seen as a header set with cardinality equal to one. A header set is expressed by a comma-separated list of protocol identifiers, enclosed in curly braces; e.g., `ip in {vlan, llc}` selects all the packets having IP directly encapsulated in VLAN or LLC. The `any` placeholder can be used to define a single encapsulation in which any protocol is valid. For instance, the header chain `tcp in any in wifi` accepts packets such as WiFi-IP-TCP and WiFi-IPv6-TCP, while WiFi-IPv6-IP-TCP is rejected because `any` matches a single protocol only. The last components of a header chain are the *repeat operators*, i.e. '`+`', '`*`' and '`?`', which mean respectively *(i)* one or more, *(ii)* zero or more and *(iii)* zero or one consecutive occurrences of one or more protocols. E.g., `tcp in ip+ in ppp` accepts any packet having TCP encapsulated in a sequence of one or more IP headers, finally encapsulated in PPP. Instead, `tcp in any+ in ppp` allows, between TCP and PPP, any protocol to be repeated any number of times, such as in the packet PPP-IP-IPv6-IP-TCP.

It is worth noting that a header chain specifies a sequence of protocols that could be everywhere in the packet, and therefore could be preceded and followed by any protocol repeated an unspecified number of times. For instance, `ipv6 in ip` does not mandate the use of a specific encapsulation at the link layer, hence all the supported ones are allowed (e.g., plain Ethernet, Ethernet with VLANs, etc.). An exception is given by the sequences having, in the right-most position, the starting protocol of the database in use; e.g. `ip in ethernet in startproto`[2] matches the packets having IP encapsulated in Ethernet, which in turn is not encapsulated in any other protocol.

III. Building the encapsulation DFA

This section presents the algorithm used to create the encapsulation DFA, i.e., a deterministic FSA that describes the traffic to be filtered according to a NetPFL header chain and the encapsulations defined in a NetPDL database. Those encapsulations can be represented with a Protocol Encapsulation Graph (PEG), a direct, potentially cyclic graph modeling the encapsulation relationships among protocols. Each node of the PEG corresponds to a different protocol, while the edge from X to Y means that, within a packet, the protocol Y could be directly encapsulated into X. In this section we refer to the PEG shown in Figure 2, excluding dashed lines and protocols.

Although the PEG looks similar to an automaton, the encapsulation DFA cannot be simply obtained by removing edges from the PEG itself. An example is provided by the filter `tcp in ip in ip in ipv6`, which requires exactly two IP headers between IPv6 and TCP, which cannot be modeled by a naive transformation of the PEG in Figure 2 into an automaton. As a consequence, a more complex algorithm for the creation of the encapsulation DFA that models arbitrary header chains is needed.

[1] Since IPv4 traffic is nowadays much more common than IPv6, in this paper we use the `ip` token to refer to IPv4.

[2] In NetPDL, `startproto` is a dummy protocol that identifies the beginning of the packet; all link-layer protocols defined in the NetPDL database are encapsulated directly into it.

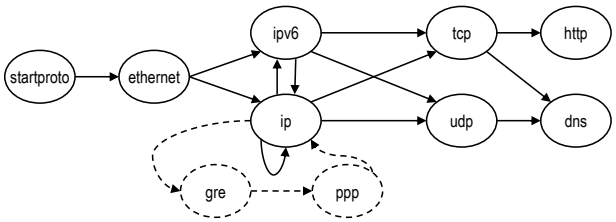

Fig. 2. Protocol Encapsulation Graph.

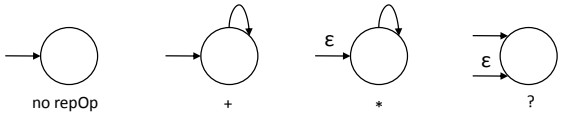

Fig. 3. Building blocks of the FSA.

A. Translating the NetPFL string to a DFA

The first step of the algorithm aims at creating a deterministic FSA (DFA) which is derived from *(i)* the NetPFL filtering string and *(ii)* the encapsulation rules defined in the given NetPDL database.

To perform the translation into a FSA, the NetPFL filtering string is split in a `target` protocol followed by an arbitrary number of `tokens` (potentially zero), where `target` is the left-most protocol of the header chain. Instead, a `token` is defined as:

```
(in|notin) pSet [repOp]
```

where `pSet` can be a header set or the `any` placeholder, and `repOp` determines how many instances of that `pSet` can be present (at most one, from zero to N, from one to N). Obviously, if the `repOp` is not specified, the `pSet` must appear exactly once. The above mentioned elements of the NetPFL string are converted, from right to left, one at a time, into automaton basic building blocks, depending on their repeat operator. The translation rules are depicted in Figure 3 and derive from the standard mapping rules defined from the FSA theory [3]. Furthermore, since the header chain allows the sequence of protocols to be everywhere into the packet, optionally preceded and followed by any protocol repeated an arbitrary number of times, the resulting automaton begins and ends with an "eatall" state, equivalent to the $.*$ element of the regular expressions[3]. Then, the last "eatall" state of the automaton is replaced with an equivalent self loop, firing with any symbol of the alphabet, over the last state of the automaton. All states are then connected in order and the rightmost one represents the accepting state of the automaton.

Further down in the algorithm we will need to remember which protocol originated each state: for this reason, each state is associated with the protocols specified by the `token` (or `target`) from which it derives[4]. If a state has an incoming ϵ transition, the state itself is also associated with the protocols related to the origin state of the transition itself. We perform this operation because, when the control of the automaton reaches the origin state, the ϵ transition causes the control to spontaneously reach the target state; hence, when parsing the packet, any protocol reached in the origin state will be reached also in the target one.

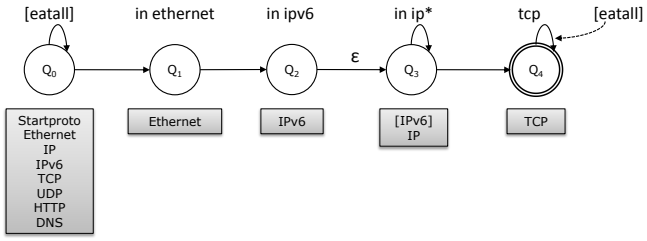

Fig. 4. FSA representing a header chain.

As an example, the FSA of Figure 4 represents the automaton built from the header chain `tcp in ip* in ipv6 in ethernet`. As highlighted with the double circle, the rightmost state represents the accepting state. Figure 4 shows also the element of the NetPFL string from which each state of the FSA derives (at the top), and the protocols associated with each state (in the grey boxes at the bottom). The IPv6 entry in the box related to state Q_3 is enclosed in square brackets to emphasize that this association is a consequence of the ϵ transition.

So far, FSA transitions are not associated with any symbol, except for the ϵ transitions that derive directly from the building blocks of Figure 3 and the self loop on the last state. In order to label properly each transition, we must define the alphabet of the FSA, which consists in the set of protocol encapsulation rules that derive from the PEG created from the NetPDL database in use. Each symbol of the alphabet is named after the two protocols involved in that encapsulation rule, the (abbreviated) name of the originating protocol first, the target last. For instance, the transition from Ethernet to IPv6 originates the symbol `eth-ipv6`, which is received by the FSA if IPv6 is directly encapsulated into Ethernet.

In our FSA building algorithm, a transition is labeled with all the symbols having the name satisfying the following constraints: *(i)* the first part, representing 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 NetPFL `token`/`target` from which the destination state derives, hence *excluding* the associations derived from the possible presence of an ϵ transition.

Figure 5 shows the result of the previous labeling rules when applied on the transition of the FSA in Figure 4. In this case, the $*$ symbol is a compact form to indicate that the transition

[3]Filters having `startproto` in the right-most position are an exception to this rule, since by definition this fictitious protocol represents the beginning of the packet. In those cases, the leading "eatall" state is omitted.

[4]In case the `pSet` is preceded by the `notin` keyword, the state is associated to all the protocols in the PEG, excluding those listed explicitly in the token.

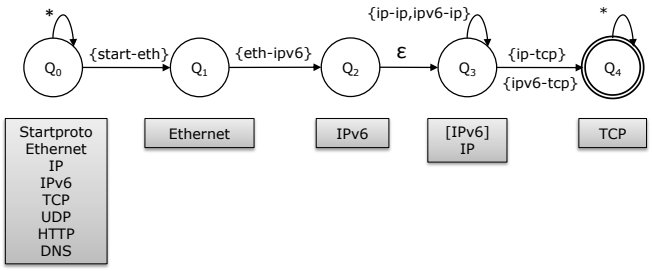

Fig. 5. FSA with labeled transitions.

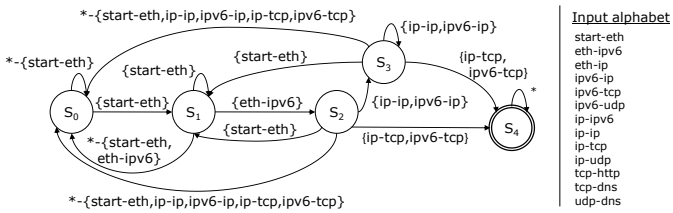

Fig. 6. DFA representing a header chain.

fires for every symbol of the alphabet.

Since the FSA created so far may be non-deterministic, it must be converted into a deterministic form using the well-known algorithms defined in the literature [3]. Figure 6 comes from the determinization of the automaton of Figure 5; the notation '$* - \{...\}$' indicates each symbol of the alphabet except those in the curly braces.

B. Assigning a single protocol to each state

While the DFA obtained so far looks nice from a theoretical point of view, it still cannot be used to generate the executable code that implements the given NetPFL filter.

In order to carry out this final lowering step, we need to associate each state of the DFA with a single protocol, so that reaching a certain state of the DFA corresponds to reaching a specific protocol within the packet under analysis. Unfortunately, automata obtained by our building process may not satisfy this condition. For instance, the original FSA in Figure 5 shows states associated with multiple protocols (e.g., states Q_0 and Q_3); the situation may become even worse in the next steps, as states originally associated with a single protocol can lose this property in the determinization process, when states are manipulated (e.g., joined or split) by FSA algorithms.

We designed an algorithm to label, whenever possible, each state with the corresponding protocols.

First, each state is inspected and, if *all* of its incoming transitions share the second part of their name (i.e., the target protocol of the encapsulation rule), then that state is labeled with that protocol. Two exceptions are *(i)* the initial state of the automaton, which corresponds to a single protocol (i.e., Startproto), only if it does not have any incoming transitions[5] and *(ii)* the accepting state, whose self loop is not considered.

[5]Note that, since a symbol leading to Startproto does not exist, an incoming transition would associate the initial state with multiple protocols.

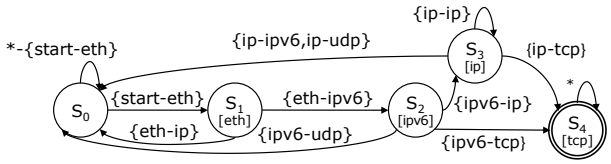

Fig. 7. DFA after the protocol assignment.

Second, unnecessary symbols are pruned from transitions, as we know (from the PEG) that, while control is a given state, the FSA can receive only symbols whose origin protocol is the one associated with the state. Symbol pruning is done by inspecting the outgoing transitions of each state: if that transition is associated with a symbol whose first part does not match the protocol associated with the state itself, that symbol is removed from the transition. Obviously, we remove all the transitions that remain without symbols and all the states that are disconnected from the rest of the FSA. These operations are repeated until there are no more changes in the DFA; the final result of this step in our example is shown in Figure 7.

Unfortunately, when the previous algorithm terminates, some states may still not be associated with a single protocol. The solution consists in transforming the obtained DFA into an equivalent form in order to reach our objective. Each *unlabeled* state is split into multiple states, one for each protocol identified by the target protocol of its incoming transitions[6]. For example, the dashed state in the left of Figure 8 originates two states, one associated with IP and the other with IPv6, as shown in the right part of the figure. A transition originating in an expanded state is replaced with new transitions, based on the origin protocols of the symbols labeling the transition itself. Each of those new transitions starts in the new state representing the source protocol of its symbols, and ends in the same state of the original transition. For example, the transition exiting from the dashed state in the left of Figure 8 originates two transitions: one labeled with `ip-ipv6` and exiting from the new state representing IP, the other firing with `ipv6-tcp` and coming from the new state associated with IPv6. Similarly, each transition ending in an expanded state is managed according with the target protocols of its symbols. In our example, the transition leading to the dashed state is replaced with two transitions, the former firing with `eth-ip` and terminated on the new state representing IP, and the latter labeled with `eth-ipv6` and entering into the new state associated with IPv6. Figure 8 shows also how the self loop on an expanded state, for each one of its symbols, originates a new transition starting and ending in the proper new states.

After the expansion some states could be useless, because their transitions originate circular paths that do no longer bring the control to an accepting state. We identify those "traps" with a reverse post-order visit of the DFA, starting from the accepting state; states that cannot be visited are removed. Figure 9 shows the encapsulation DFA recognizing

[6]The initial state is also associated with `startproto`.

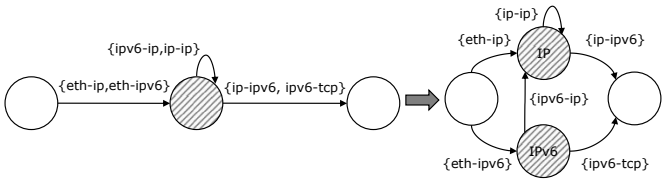

Fig. 8. Expansion of a state and the related transitions.

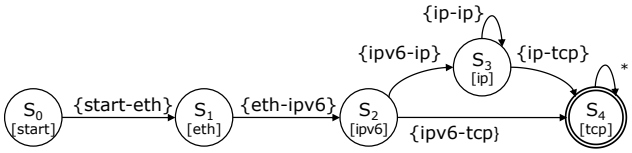

Fig. 9. Final DFA implementing the NetPFL filter of our example.

packets that match the filter `tcp in ip* in ipv6 in ethernet`.

Finally, the enpcasulation DFA is transformed into a piece of executable code that implements the filter, which is then used to analyze network packets.

It is worth remembering that a filter could have more header chains, composed through the Boolean operators `and` and `or`. In this case, our algorithm is executed for each header chain, and the resulting DFA are combined using traditional algorithms defined in literature.

IV. Implementation

The proposed algorithm was implemented in the NetBee library [4]. This library includes a `tcpdump`-compatible tool named `nbeedump` for packet filtering, which exploits the NetVM [5] virtual machine for executing the packet filtering code. The overall system architecture is shown in Figure 10: `nbeedump` receives as input both the NetPDL database (containing the format of the supported protocols and the encapsulation rules) and the NetPFL filtering expression. These information are taken by a compiler that, after the application of several optimization algorithms, emits the final filtering code under the form of NetIL instructions (i.e., the NetVM assembly language). The NetIL can be interpreted by the NetVM itself, or transformed into native code for a bunch of target architectures. Within the above compiler, the algorithm presented in this paper is implemented in the *DFA builder* module, which takes the PEG (dynamically extracted by the NetPDL database) and the NetPFL filter, and builds the DFA representing the packet filter. Subsequently, the *DFA lowering* module generates the corresponding NetIL code. Input symbols of the FSA are generated by the *protocol scanner*; even if it is a logically separated module, its operations are in fact performed by the same assembly program implementing the DFA. This means that, when generating the NetIL code for a state, we link together both the code that implements the automaton and the one that handles the encapsulations.

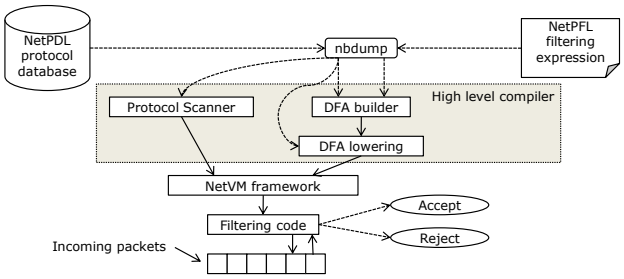

Fig. 10. Overview of the building blocks to generate filtering code.

#	NetPFL Filter
1	`tcp in ip`
2	`tcp in ip in ethernet`
3	`tcp in ip in ethernet in startproto`
4	`tcp in any+ in ethernet`
5	`http notin {tcp,udp}`
6	`tcp`

TABLE I
NetPFL filters used in the tests.

V. Experimental results

The algorithm presented in this paper has been validated with two types of tests: the former evaluates the time required to create the FSA and generate the machine depended code that recognizes packets matching the filter, while the latter addresses the performance of the generated code at run-time. Tests have been performed on a workstation with 4 GiB of memory, CPU Intel E8400 @ 3.00 GHz and OS Ubuntu 10.04, kernel 2.6.32-38-generic, 64 bits. All tests were executed with the `nbeedump` tool, which used the NetPFL filters shown in Table I and the NetPDL protocol database shown in Figure 2 (including dashed lines and protocols[7]).

It is worth noting that, at the best of the authors' knowledge, no other filtering languages exist that allow users to specify filters based on protocols encapsulations. For instance, neither `libpcap` [6], which represents the foundation of many packet filtering tools (e.g., `tcpdump`, Wireshark), nor the display filters [7] implemented in Wireshark (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 cannot 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 proved to be at least equivalent to the state of the art in this field [8].

A. Compilation time

This test evaluates the filter compilation time, i.e. the time required for generating the actual x64 assembly code

[7]Actually, the original NetPDL database available on the nbee.org web site accounts for more than one hundred protocols; we used a reduced database for the sake of clarity.

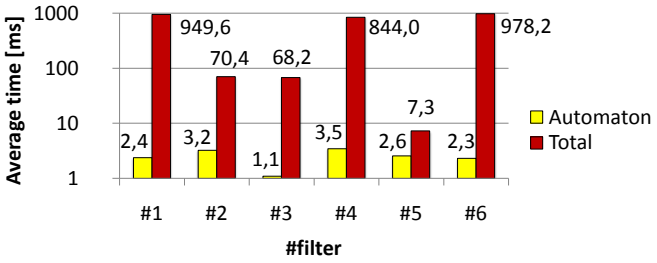

Fig. 11. Performance of the code generator.

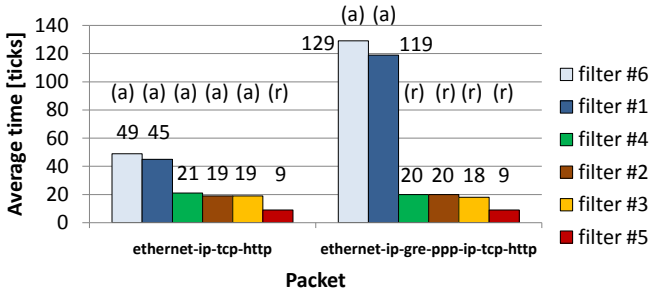

Fig. 12. Performance of the generated filters.

implementing a specific NetPFL filter. 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. Our test measures the time spent by our algorithm with respect to the total code generation time. Each filter has been compiled thousand times and averaged, obtaining the numbers depicted in Figure 11. Results show that the time required for creating the encapsulation DFA is negligible compared to the total generation time, for all the considered filters. Furthermore, the time needed to build the FSA increases when the number of protocols that could match the initial *eatall* state in the FSA representing the filter is reduced. The reason is that our algorithm expands the initial state in most of the protocols defined in the database in use, and then prunes the unnecessary ones. As a consequence, NetPFL filters that explicitly mention `startproto` generate very compact filtering DFA and represent the fastest generation case for our algorithm.

B. Filtering time

This test aims at evaluating the quality of the resulting filtering code (i.e. the x64 assembly program) executed on a set of real packets. The filtering code measures only the time needed to check if a packet satisfies the filter, without taking into account additional overhead such as reading packets from disk (or from the NIC) and more. Each filter has been repeated one thousand times on each test packet and the results have been averaged. Since a filter execution lasts a few nanoseconds, measurement has been performed with the RDTSC instruction available in the Intel x64 instruction set.

Figure 12 shows the number of CPU ticks needed for the filters of Table I applied to a first, simple packet, and a second with a tunnel involving the IP protocol; precise encapsulations are written on the figure itself. As evident, the cost of an accepting filter (shown with the `(a)` on top of the bar in Figure 12) decreases when the filter is more specific, i.e. leaves less freedom to the protocols that may appear in a certain position of the packet. Furthermore, accepting filters will require more time to analyze the complex packet because the encapsulation sequence of interest is matched later within the packet itself. Finally, filter #5 is so fast because it represents a condition that, according with the PEG in use, does not match any of our packets, which are then discarded almost immediately after a few checks.

VI. Conclusions

This paper presents an algorithm that enables the creation of efficient packet filters operating on protocol encapsulations. Our algorithm enables the creation of more specific filters based on the header chains defined in the NetPFL filtering string. This capability is important for many network tools based on packet filtering capabilities that will be deployed in the near future, as it would be needed to support multiple encapsulations; it will also enable the efficient filtering of the traffic we want, given the growing amount of packets that use very complex encapsulations.

Although we cannot compare our performance with other systems (as we are not aware of other software supporting filtering on protocol encapsulations), our preliminary results seem to confirm that the generated filters can be extremely efficient, suggesting that our NetBee library could be used as a foundation for more sophisticated packet capturing tools that require fast and flexible traffic filtering.

Future works include the extension of our algorithm to support other features defined in the NetPFL language, such as the capability to specify rules based on a specific instance of a protocol. The filter `tcp in ip%2`, which matches when the `ip-tcp` encapsulation refers to the second instance of the IP protocol, represents a possible example.

References

[1] F. Risso and M. Baldi, "Netpdl: an extensible xml-based language for packet header description," *Comput. Netw.*, vol. 50, no. 5, pp. 688–706, Apr. 2006.

[2] L. Ciminiera, M. Leogrande, J. Liu, F. Risso, and O. Morandi, "A tunnel-aware language for network packet filtering," in *Proceedings of the Global Communications Conference (Globecom 2010), Miami, Florida, USA*. IEEE, Dec 2010, pp. 1–6.

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

[4] F. R. et al., "The netbee library," Nov 2007, version 0.2. [Online]. Available: http://www.nbee.org

[5] O. Morandi, F. Risso, P. Rolando, S. Valenti, and P. Veglia, "Creating portable and efficient packet processing applications," *Design Automation for Embedded Systems*, vol. 15, no. 1, pp. 51–85, Mar. 2011.

[6] S. McCanne and 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. C. et al., "Wireshark display filters," Oct 2008. [Online]. Available: http://wiki.wireshark.org/DisplayFilters

[8] O. Morandi, F. Risso, M. Baldi, and A. Baldini, "Enabling flexible packet filtering through dynamic code generation," in *Proceedings of IEEE International Conference on Communications (ICC 2008), Beijing, China*, May 2008, pp. 5849–5856.