Modeling Complex Packet Filters with Finite State Automata

Modeling Complex Packet Filters with Finite State Automata

Marco Leogrande, *Member, IEEE,* Fulvio Risso, *Member, IEEE,* and Luigi Ciminiera, *Member, IEEE*

***Abstract*—Designing an efficient and scalable packet filter for modern computer networks becomes each day more challenging: faster link speeds, the steady increase in the number of encapsulation rules (e.g., tunneling) and the necessity to precisely isolate a given subset of traffic cause filtering expressions to become more complex than in the past. Most of current packet filtering mechanisms cannot deal with those requirements because their optimization algorithms either cannot scale with the increased size of the filtering code, or exploit simple domain-specific optimizations that cannot guarantee to operate properly in case of complex filters. This paper presents pFSA, a new model that transforms packet filters into Finite State Automata and guarantees the optimal number of checks on the packet, also in case of multiple filters composition, hence enabling efficiency and scalability without sacrificing filtering computation time.**

***Index Terms*—pFSA, Packet filters, Finite State Automata.**

I. Introduction

In the recent years, we witnessed many changes in the computing world. The network interaction among devices has evolved significantly, so we had to re-engineer our computer networks in order to accommodate many new use cases. A large portion of these requirements introduced new features in the network protocols stack, whose complexity increased as a consequence. For instance, low-level network protocols are growing in number: new solutions, arising in particular for the purpose of network virtualization (e.g., 802QinQ, VXLAN), are rapidly transforming our Ethernet frames [1]. The middle layers of the protocol stack are facing a similar metamorphosis: examples include the widespread adoption of Virtual Private Networks with their bizarre tunneling mechanisms, the necessity to transport IPv6 traffic over IPv4 networks (with different encapsulation methods, such as pure IPv6 encapsulation in IPv4, or through GRE or even UDP, and more) and WAN traffic transports.

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

This paper presents **pFSA**, a new model for packet filtering that ensures the optimal number of checks on the packet in order to take the matching/not-matching decision. This result is obtained by transforming packet filtering rules into Finite State Automata (FSA), which guarantee optimal results even in case of multiple filters combined together. Vice versa, the *ad hoc* optimization techniques used by most of the previous approaches are based on heuristics that cannot provide such guarantees, which are needed to ensure the best performance when operating in the conditions mentioned before (multiple filters or unconventional encapsulations). Furthermore, our model is generic enough so that it does not require *a priori* protocol definitions: in our prototype the protocol database is provided at run-time and it can be easily extended or modified in order to recognize any protocol or encapsulation the user is interested in. This means that we can create the best filtering automaton whatever encapsulation we may have, including unusual protocol patterns such as tunneling (and self-tunneling) of any kind; the generated filter is able to locate the desired pattern in the network traffic, independently from the actual protocol stack of the given packet. Finally, we present also a prototype implementation that translates the pFSA model into running code, although this step cannot formally maintain the optimality properties guaranteed by the model.

This paper is structured as follows. Section II presents the state of the art; Section III introduces the proposed model; Section IV presents the application of that model to the packet filtering domain, while Section V is dedicated to the problem of optimization on protocol fields. Finally, Section VI presents an overview of our implementation, leaving the experimental evaluation to Section VII and conclusions to Section VIII.

II. Related Work

The **CMU/Stanford Packet Filter** [2] (CSPF) represents the ancestor of any modern packet filter. It introduced the concept of a kernel-level virtual machine that executes an application-provided program (i.e., the packet filter), which can be defined at run-time. However, its optimizations capabilities were limited.

The **Berkeley Packet Filter** [3] (BPF) is also based on a virtual machine and brings some notable improvements, such as the adoption of the *Control Flow Graph* model, which enables the deployment of compiler techniques to remove redundant checks from the generated code. The BPF model was later improved by **BPF+** [4], which uses even more aggressive optimizations derived from software compilation techniques and adds a Just-In-Time (JIT) compiler.

PathFinder [5] adds the possibility to compact the Control Flow Graphs of different filters. Each expression in a filter is exploded into a list of *cells*, each one describing a step in

Marco Leogrande, Fulvio Risso and Luigi Ciminiera are with the Department of Computer and Control Engineering, Politecnico di Torino, Torino, 10129 Italy, e-mail: {marco.leogrande,fulvio.risso,luigi.ciminiera}@polito.it.

Manuscript received June 06, 2012; revised November 10, 2013.

DOI: 10.1109/TNET.2013.2290739

the construction of the final check; equivalent cells coming from multiple filters *may* be merged together. However, filters are optimized only if they share a common prefix; for instance, `tcp.sport` is always checked twice in the expression `(tcp.sport == X and tcp.dport == W) or (ip.src == K and tcp.sport == Y)`.

The **Dynamic Packet Filter** [6] (DPF) extends the previous approach by introducing the capability to generate native code instead of running the filter into an interpreter. Furthermore, field coalescing is introduced, allowing fields at contiguous offsets to be checked together; for example a single 32-bits check against the word `0x00800090` is performed for the filter `tcp.sport == 0x80 and tcp.dport == 0x90`.

The recently proposed **Stateless FSA-based Packet Filter** (SPAF) [7] exploits Finite State Automata for packet filter generation and guarantees, by construction, code optimality and safety. Each protocol is modeled through a byte-consuming automaton, which reads the bytes that are part of the protocol and follows the encapsulation rules (e.g., the starting state of the `IP` protocol is linked to the exit state of the `Ethernet` protocol when the bytes associated with the `EtherType` field have the proper value); different automata are then joined together using the algorithms known from the literature. However, SPAF is extremely slow in the automata generation phase, because the protocol field abstraction is lost very early in the computation, hence the amount of generated states tends to be rather high. This has a huge impact on FSA construction, as determinization (required in FSA composition) is exponential in the number of states. For this reason, SPAF is appropriate only for applications that can tolerate rather long filter generation times.

Swift [8] focuses on packet filtering updates in strict real-time. The ultimate goal is to add a new filtering rule for a TCP session as soon as its three-way handshake is completed, which is done through a tree-like structure similar to PathFinder. This enables also the use of new x86 *SIMD* instructions to perform multiple checks in parallel.

Ruler [9] is a packet rewriter designed to anonymize traffic traces, which can also be used for packet filtering. It introduces a flexible high-level language for deep packet inspection and rewriting, which is mapped on an extension of the FSA model. Since it is based on automata, Ruler shares a degree of similarity with pFSA and SPAF, but its design goals are sufficiently different to produce noticeably distinct results; furthermore, its source language is not general enough to specify complex filter statements or certain commonly encountered protocol structures, such as IPv6 extension headers.

To the best of our knowledge, SPAF is the only packet filter model that uses a FSA-like approach. If we broaden the area of research, we find that only a handful of publications has proposed extensions to the base FSA formalism that might be similar to ours. **pfsr** [10] is a predicate-augmented finite state recognizer, that aims at simplifying the Finite State Automata used in natural language processing. Even if the authors describe in detail their model extension, providing definitions and algorithms, the scope of the predicate that they introduce is quite different from ours, as it is used only to define arbitrary sets of input symbols. **EFSA** [11] models a fast intrusion detection and prevention system by making use of augmented FSA transitions with arbitrary predicates. The EFSA paper, however, does not describe predicates in detail: e.g, predicate optimization, that is a critical issue in packet filtering, is not mentioned at all. **XFA** [12] is also based on an augmented FSA, but states are associated with a generic executable code for efficient pattern matching, which is not appropriate for optimizing predicates in packet filtering applications.

Other packet filtering technologies such as **FFPF** [13] are not described in detail here, as they aim to solve orthogonal problems, such as how to multiplex incoming packets between different packet filters, but do not offer any improvement to the filtering model itself.

The most common filtering architectures (excluding SPAF) tend to rely on *ad hoc* optimizations, often inspired at compiler-oriented techniques, which are applied on the code that has to be executed. Some of them exploit optimizations to coalesce packet accesses or use hardware-efficient assembly instructions. However, no guarantees of optimality can be given; furthermore, many of those optimization algorithms scale exponentially with the number of instructions of the generated filter, which becomes a major problem when the size of the filter grows because of more complex conditions or uncommon encapsulations, including tunneling. Instead, pFSA defines a packet filtering model based on the FSA formalism that guarantees optimal filtering construction (by minimizing the number of checks on the packet) and that overcomes the SPAF limitation in terms of compilation time. This is due to the capability to derive the FSA from the protocol abstraction, while SPAF adopts a byte-stream approach, that generates a much more verbose automaton compared to our pFSA.

III. Finite State Automata with Predicates

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

A. Definition of pFSA

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

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

where:

- $\mathbf{Q}$ is a finite set of *states*;
- $\mathbf{\Sigma}$ is the set of *input symbols*;
- $\delta_\mathbf{p}$ is a *transition function with predicates* (described below);
- $\mathbf{q_0}$ is the *starting state*, among those in Q;
- $\mathbf{F}$ is a set of *accepting states*, among those in Q.

A **transition function with predicates** mimics the meaning of "classic" transitions, but adds the possibility to tune the transition behavior according to a set of **Boolean predicates**,

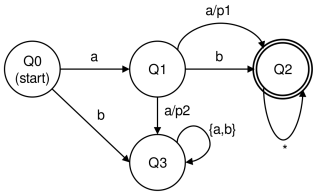


Figure 1. pFSA example.

whose semantic is orthogonal to the input symbols of the automaton. It is defined as:

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

where:

- $\mathbf{q_1}$ is the state from which the transition takes place;
- σ is the input symbol that triggers the transition, or the special value ϵ (*epsilon*) if no input symbol should be consumed;
- $\mathbf{p}$ is a Boolean predicate that "activates" the transition, that is allowed to fire only if the predicate is *true*;
- $\mathbf{q_2}$ is the state reached by the current transition.

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

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

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

We do not pose any particular limitation to the predicates; however, the *predicate evaluator* cannot change the internal state of the automaton, such as move the current state from `Qn` to `Qm`, or change the input string, etc. In other words, the predicate evaluation step must have no side effects other than returning the current Boolean value of the predicate: it is duty of the automaton itself to interpret the returned value and act accordingly. Given this limitation, a predicate could be as simple as *"is the value of variable $x odd?"*, but even a question like *"is IP multicast enabled on the `eth0` network interface?"* is valid, as long as it is possible to give a *true/false* answer. Predicate values are allowed to change only when a new input symbol is consumed; in other words, they are frozen when an ϵ-transition is going to fire. This restriction is needed for the general algorithms to work, but does not have any impact on our usage of the model. More details will be given in Section IV-C, after we describe how we use the model to filter network packets, together with some examples (Figures 7 and 8).

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

B. Running a pFSA

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

Whenever, according to the current state and the input symbol received, a p-transition should be activated, its predicate is inspected and its current Boolean value is returned; that transition fires if the predicate is found to be *true*, otherwise another transition is taken. Note that if multiple p-transitions have the same start state and are labeled with the same input symbol, a subset of them (from zero to all) might fire at the same time, according to the values taken at run-time by their predicates: this is an important issue to consider when stating whether a pFSA is deterministic or not (more details in Section III-C). Referring to Figure 1, if the symbol `a` is received when the control is in state `Q1`, two p-transitions might fire: the one that leads to state `Q2` can fire depending on the value of predicate `p1`, while the one that leads to state `Q3` can fire according to the value of predicate `p2`. Only the run-time values of `p1` and `p2` can clarify which state(s) will be reached: either `Q2`, or `Q3`, or both of them or none.

C. Determinism

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

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

While this definition looks simple, stating whether a pFSA is deterministic or not may be complicated in practice, because the outcome depends on the values of the predicates, that can be evaluated only at run-time. For instance, if two transitions labeled with `p1` and `p2` exit from the same state and are associated with the same input symbol (such as in Figure 1), that pFSA is possibly non-deterministic, as both transitions might be enabled at the same time. Conversely, if those transitions are labeled with `p1` and $\overline{\texttt{p1}}$ there is no determinism issue, as the logic rules assert that exactly one between those predicates is true at any instant. Consequently, if in a

given pFSA no state has multiple transitions with the same symbol (i.e., the base FSA is deterministic) and the predicates associated with the transitions exiting from every state, labeled with the same input symbols, are only in the form `p1` and $\overline{\texttt{p1}}$, that pFSA is still deterministic.

D. Algorithms

One of the main advantages of reusing the FSA formalism is that many definitions, algorithms and optimizations from the literature (e.g. [14]) can be reused with little effort.

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

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

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

Predicates Cartesian product: It is used to determinize a pFSA in which a state has multiple outgoing transitions, all triggered by the same input symbol but associated with different predicates[3], such as in the leftmost part of the fully specified pFSA in Figure 2. To guarantee the determinism property, the pFSA is determinized by introducing a number of transitions that is equal to the Cartesian product of the existing predicates; refer to the central part of Figure 2, where each transition is terminated on the state that would be activated in the original pFSA, or on a new state (e.g. `Q12`) that captures multiple states of the original automaton. This way we can guarantee that only one out of the four transitions leaving from `Q0` for input symbol `a` can be true, independently from the actual values of predicates `p1` and `p2` at runtime. The resulting pFSA can be further optimized by additional algorithms, such as compaction of indistinguishable states: e.g., in the automaton in the rightmost part of Figure 2, state `Q12` has been merged with `Q1`.

Predicate anticipation: Sometimes, the pFSA determinization algorithm may move a predicate bound to an ϵ-transition over another transition that precedes it, with some additional adjustments; this is useful when the preceding transition is not already associated with a predicate, and the anticipation allows a state simplification. This transformation is possible because it is guaranteed that the predicate has the same Boolean value in both cases: as already outlined in Section III-A, predicate values are allowed to change only when a new input symbol is consumed. A simple example is shown in Figure 3, where predicate `p1` on the ϵ-transition between states `Q2` and `Q3` is moved over the previous transition from `Q1` to `Q2`. However, moving the predicate requires the creation of two transitions (one labeled as `a/p1`, the other as `a/`$\overline{\texttt{p1}}$) and the duplication of state `Q2` (second step of Figure 3). The final pFSA, obtained by removing the ϵ-transition and by compacting the states that are indistinguishable (i.e., `Q2` and `Q3`, and `Q2'` and `Q4`), is depicted in the rightmost part of Figure 3.

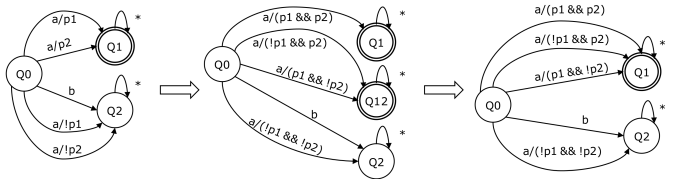


Figure 2. pFSA predicates Cartesian product.

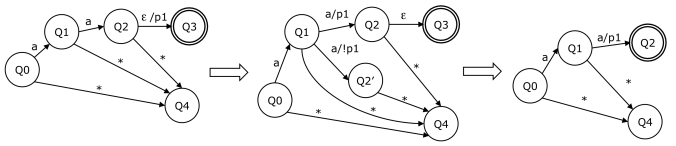


Figure 3. pFSA predicates anticipation.

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

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

The **ϵ-closure** of a set of states recursively adds to the set of states S all those that are reachable, through an ϵ-transition, from any state already in S. If some predicates are found on those ϵ-transitions, their Cartesian product is computed, possibly also against the predicates already discovered in the previous step.

Finally, the modified **transition function $\boldsymbol{\delta_p}$** is used in the state compaction step to detect if two states are equivalent and can be merged together. To achieve this, the transition function δ_p checks whether, upon the receipt of a given input symbol σ, two transitions exist that lead from the couple of states

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

[3]Starting from Figure 2, the notation `!predicate` (borrowed from popular programming languages) is used to express a negated condition.

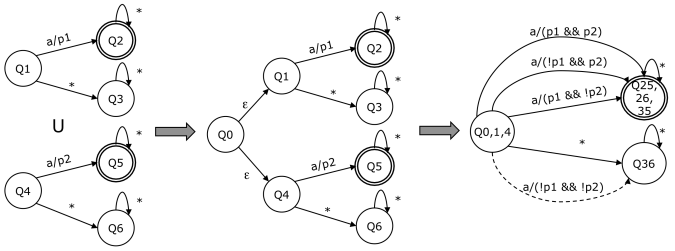


Figure 4. Example of a pFSA with complex predicates: the *union* case.

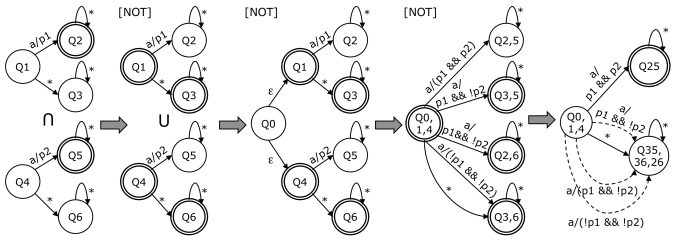


Figure 5. Example of a pFSA with complex predicates: the *intersection* case.

under testing to any other couple of states that were already found distinguishable. In a pFSA, if a predicate is present over a transition, then, for the distinguishability test, all the other transitions that are associated with the same input symbol and exiting from the same state must be considered as well.

E. Predicates composition

The algorithms presented in the previous section are used to combine together more pFSA, leading to a new, equivalent pFSA that retains all the properties guaranteed by the pFSA formalism. The example in Figure 4 shows the union of two simple pFSA through the required processing steps: a new state is connected to the original pFSA through two ϵ-transitions (in the middle), then the final pFSA that comes after determinization and minimization is shown at the right. The example in Figure 5 looks more complicated as it shows the intersection between two pFSA, which occurs by transforming that operation into a set of union and negation steps[4].

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

[4]In Figures 4 and 5, transitions with dashed lines are redundant and may be deleted, as they are included in the default `'*'` arc.

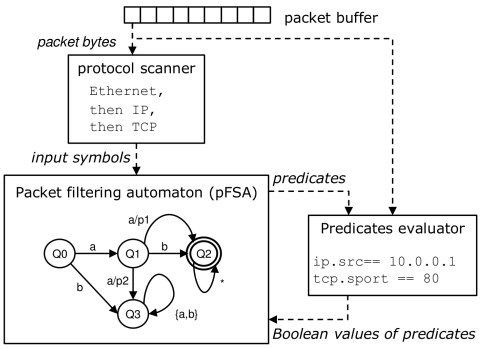


Figure 6. Overview of the system in which pFSA are used for packet filtering.

IV. pFSA FOR PACKET FILTERING

Although the pFSA model is rather general and can be adapted to different contexts, this paper focuses on its application in packet filtering and shows how multiple filters can be combined together with a solid guarantee of optimality in terms of number of checks on the packet. This section focuses on this objective, presenting how the pFSA model can be used to describe a generic filter, exploiting pFSA properties to reduce (and optimize) complex filtering expressions. To do so, we should be able to translate a filtering expression into an equivalent pFSA, so that: *(i)* if a packet matching the provided filter is given to our system, the pFSA should end in an accepting state; *(ii)* otherwise, if the packet does not match, the pFSA should end in a non-accepting state.

We will describe how the packet filtering machinery is mapped in the pFSA model: namely, how states, symbols and predicates are defined. An overview of the system is given in Figure 6, while a detailed view of each block will be given in Section IV-E.

A. States

The initial construction of the pFSA associates each state with a network protocol, that represents the protocol that has been reached while scanning the current packet[5]. For instance, when the `ip` state becomes active, it means that the `IP` protocol has been found in the current packet and that the protocol scanner is going to read the first byte associated with it. As a consequence, for simple pFSA, the set of accepting states includes only those states that match the protocol requested by the filter: for instance, in the pFSA modeling the filter that selects only `ip` traffic (e.g., in Figure 7), the state labeled `ip` would be the only accepting state. More details about the important bonding between states and network protocols will be presented in Section IV-D.

B. Input symbols

In a pFSA for packet filtering, each input symbol represents a single encapsulation rule, i.e., a sort of "jump" from a protocol to the next. For instance, the symbol `ethernet-to-ip` is associated with a transition that goes from a state that

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

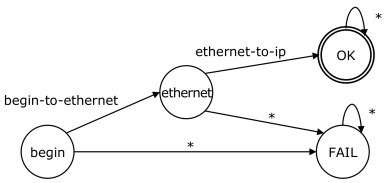


Figure 7. Example of a pFSA for the filter `ip`.

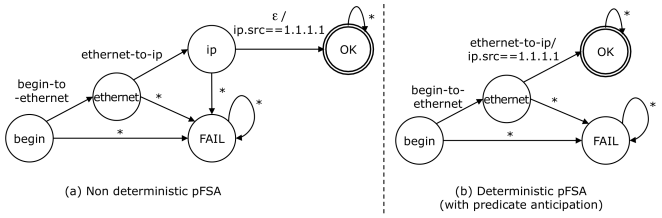


Figure 8. Example of a pFSA for the filter `ip.src == 1.1.1.1`.

represents the `Ethernet` protocol to another that represents `IP`, as shown in Figure 7. If the pFSA receives this symbol at runtime, it means that the packet currently under examination contains an instance of `IP` directly encapsulated inside an `Ethernet` header[6].

In our system, input symbols are generated by a separate module (the **protocol scanner** of Figure 6), which inspects the incoming packet, analyzes the protocols in it, generates the symbols and passes them to the pFSA engine; for each encapsulation found in the current packet, a new input symbol is generated. For instance, if a packet contains, in this order, the `Ethernet`, `IP` and `TCP` headers, then three input symbols are passed to the pFSA: `begin-to-ethernet`, `ethernet-to-ip` and `ip-to-tcp`.

The input symbols that the pFSA expects to receive (the Σ alphabet) are derived from a *protocol database*, that is provided to the engine that builds the pFSA at filter compilation time. More details about the protocol database and the building process are in Section IV-E.

C. Predicates

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

Figure 8(a) shows a simple pFSA for the filter `ip.src == 1.1.1.1`. Note that, differently from Figure 7, the `ip` state is no longer accepting: the `ip` state is connected to the actual accepting state through an ϵ-transition with the `ip.src == 1.1.1.1` predicate. If this predicate is found to be *false* at run-time (because the IP source address does not match the value `1.1.1.1`), then the path towards the accepting state is effectively barred, therefore rejecting the packet. Figure 8(b) shows the same pFSA after running the predicate anticipation algorithm, which transforms the pFSA into a deterministic automaton. It is worth noting that the two forms (a) and (b) of the given pFSA are completely equivalent and it is possible to transform one into the other, if needed.

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

The actual Boolean value of the predicates is evaluated by a dedicated module (the **predicates evaluator** of Figure 6), that is logically separated from the protocol scanner. Whenever the pFSA encounters a predicate at runtime, the evaluator is invoked and the current Boolean value of that predicate is returned.

It is worth remembering that predicates can be evaluated only when the corresponding transition is about to fire and cannot be precomputed, because their value might change every time a new input symbol is consumed. For instance, the Boolean evaluation of `ip.src == 1.1.1.1` may result in different values when filtering a packet that contains a `ip-in-ip` tunnel, depending on whether we are operating on the inner or outer `IP` header. This case is not handled in Figure 8 to keep the example simpler.

D. States and network protocols

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

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

This nice property of strong relation between states and protocols can be apparently lost when some optimizations (particularly, predicate anticipation) come into play. For instance, Figure 10 presents an example in which an intermediate state is associated with the reachability of a given protocol field, i.e., the pFSA reaches field `ip.src`. However, this does not represent a problem as input symbols (which represent

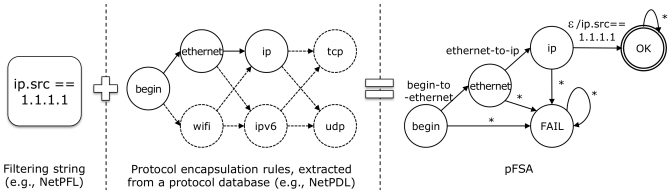


Figure 9. Building steps for a simple pFSA.

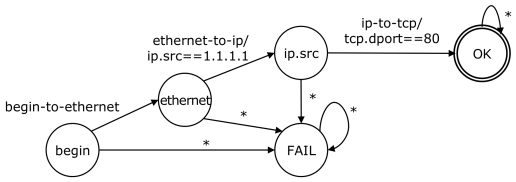


Figure 10. Example of a deterministic pFSA for the filter `ip.src == 1.1.1.1 and tcp.dport == 80`.

protocol encapsulation rules) guarantee that two states can be merged only if they are reached through the same encapsulation rule.

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

E. Building a pFSA for packet filtering

The process that creates the pFSA that represents a given packet filter involves different components, as presented in Figure 9. The packet filtering pFSA is the result of the combination of the **filtering string**, which represents the actual filtering statement, and a **protocol database**, which features a description of the protocols in terms of fields and encapsulation rules (although in this step only the latter is considered). Encapsulation rules specify how protocols are encapsulated one into the other, resulting into a directed, potentially cyclic, graph. For instance, there will be an entry that states that the `IP` protocol can be found inside `Ethernet`, but there will be no entry for `TCP` inside `Ethernet`. The protocol database should also mark the protocols that can be found at the beginning of a packet (i.e., link-layers such as `Ethernet` or `WiFi`), in order to highlight which protocols represent some sort of "starting nodes" of the encapsulation graph; in our implementation those link-layer protocols follow a fictitious "begin" protocol. The pFSA model is agnostic with respect to the protocol database, as long as it includes the required information; in fact, the choice of this external component is under the responsibility of the specific pFSA implementation.

The first step towards the pFSA creation is parsing the *filtering string* itself, splitting it in basic tokens, i.e., statements that express a condition operating on a single protocol or a protocol field, chained together with Boolean operators. A distinct pFSA is generated for each of these blocks, which are combined together using the algorithms presented in Section III-D, therefore obtaining the final pFSA. As each portion of the tokenized filtering string refers to a single protocol, it is used to traverse the encapsulation graph and to select all the paths that connect the *starting protocol* (that represents the starting state of the automaton) to that protocol. For instance, all paths that result useless for the given filter are discarded in this step. All nodes and edges selected are then used to build the pFSA, transforming each encapsulation into a possible input symbol for the automaton. An additional state is created, representing the non-accepting condition (i.e., when the packet does not match the filter); every other state is then connected to this failure state using a *star* transition[7].

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

The examples shown in Figure 7 and 8 were created with this algorithm: in both cases the `begin` state corresponds to the *starting protocol* in the above description. Those examples show also that the FSA creation process can lead to non-deterministic pFSA, such as in Figure 8.

Figure 10 represents a more complex example: a pFSA already determinized for filter `ip.src == 1.1.1.1 and tcp.dport == 80`. The filter appears optimal in the number of tests: only one path leads to state `OK`, which includes the verification of both conditions present in the filter. If the first test fails, the failure state is reached immediately, ignoring the run-time value of the second predicate.

V. PREDICATES OPTIMIZATION

The optimization of filtering predicates represents a critical issue in order to effectively model packet filters; in particular, some applications may be extremely sensible to the problem of predicate composition previously introduced in Section III-E. In fact, a model that guarantees optimality with respect to protocol encapsulations is still not enough for those applications that require complex filtering expressions operating on protocol fields: these are somewhat "outside" the pFSA model and hence, so far, are not optimized at all.

For instance, Figure 11 represents a deterministic pFSA modeling the filter `ip.src == 1.1.1.1 or ip.dst == 2.2.2.2`, which is then translated into a pFSA that requires the analysis of four predicates when the `ethernet-to-ip` input symbol is received. This exponential explosion in the number of transitions might be troublesome in complex (but very common) packet filters, e.g., those that account hundreds of tests over the same protocol fields, such as Access Control Lists operating on IP addresses.

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

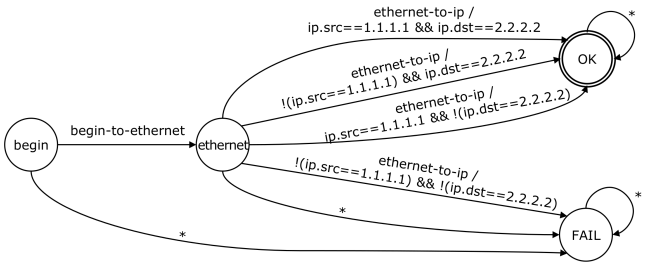


Figure 11. Example of a deterministic pFSA for the filter `ip.src == 1.1.1.1 or ip.dst == 2.2.2.2`.

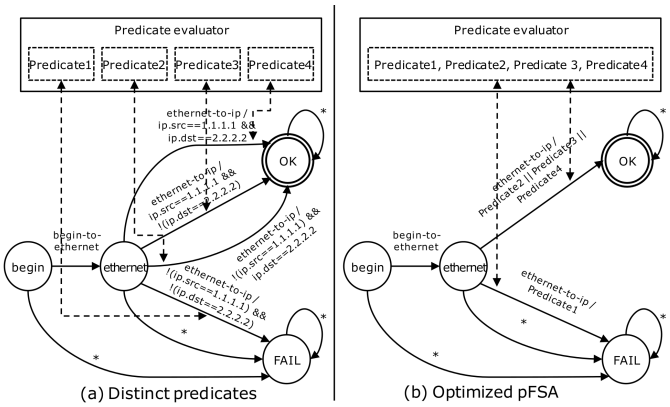


Figure 12. The main idea behind the "multilevel" implementation feature. Predicates are merged within the same predicate evaluation block, leading to a simplification of the base pFSA: multiple transitions are merged together, paving the way for further optimizations at the predicate level.

All those transitions must be evaluated by the predicates evaluator in order to determine which one will fire (if any), hence posing a substantial run-time overhead when trying to resolve the current Boolean value of multiple, arbitrarily complex predicates.

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

A. Overview

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

For the first step we created a block that can merge multiple queries coming from different transitions, instead of having different predicate evaluators for each expression such as in Figure 12(a), which is possible because the pFSA model does not mandate the internal architecture of the predicate evaluator. If predicates operate on the same protocol field (which is rather common), their evaluation is potentially faster.

The second step (shown in Figure 12(b)) simplifies the layout of the pFSA when possible. For instance, the three transitions between states `ethernet` and `OK` can be compacted into one, associated with the logical `or` of the three predicates, thus enabling further optimizations in the next step.

The third step minimizes the operations needed to evaluate the expressions by restructuring the internals of the predicate evaluator. For instance, given the predicates in Figure 12, we can structure the predicate evaluator so that the condition `ip.src == 1.1.1.1` is checked once and then its result is reused for all expressions; or, the test on `ip.dst` is not performed if its value does not change the final result.

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

B. Going multilevel: the protoFSA

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

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

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

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

dedicated to predicates optimization, where:

- $\mathbf{S}$ is a finite set of *states*, associated with the protocol fields referenced by the predicates;
- $\mathbf{\Sigma}$ is the set of *input symbols*, which consists in the union of the set of values syntactically valid for each protocol field (e.g., a predicate operating on an IP address and on the IP TOS byte originates $2^{32} + 2^8$ possible input symbols);
- δ is the *transition function*, which takes into account whether a condition on a specific field triggers the analysis of a subsequent condition on another field;
- $\mathbf{s_0}$ is the *starting state*;

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

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

C. Building a protoFSA

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

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

Each basic block is translated into a minimal FSA in which the protocol field is associated with a state, while the space of its possible values is used to define the transitions to the `OK` and `FAIL` states[9]. If, based on the protocol fields ordering mentioned above, a basic block refers to a field other than the first one, a set of states referring to its "preceding" fields is pre-pended to the state associated with the state itself. In other words, state s_i, associated with predicate P_i, is preceded by states $s_1 \dots s_{i-1}$, associated with predicates $P_1 \dots P_{i-1}$. Preceding states selected this way are connected with a default transition, such as in the second basic block (bottom left) in Figure 13. The `OK` and `FAIL` states are then associated with the transitions (in the base pFSA) that originate the current query to the predicate level.

The next step consists in building the whole protoFSA, by merging together all predicates that result from the same Cartesian product. The final protoFSA has as many final states as the number of predicates resulting from the Cartesian product, each own mapped to a p-transition of the base pFSA. However, multiple p-transitions in the pFSA can be merged together if their ending states are not distinguishable (in terms of the minimization algorithm), so we have a chance to optimize again the protoFSA through the well-known FSA composition and optimization algorithms. For instance, in Figure 12(a), predicates P_2, P_3 and P_4 lead to the same state; consequently the protoFSA can be further optimized, resulting in the final form shown in Figure 14.

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

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

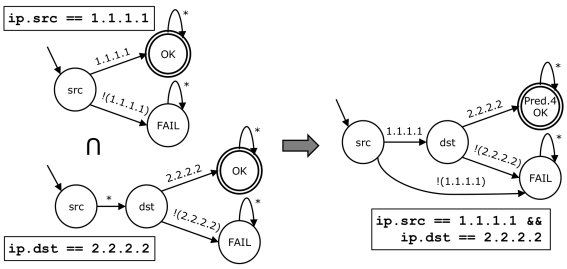


Figure 13. Example of composition of the predicate `ip.src == 1.1.1.1 and ip.dst == 2.2.2.2`, corresponding to predicate P_4 in Figure 12.

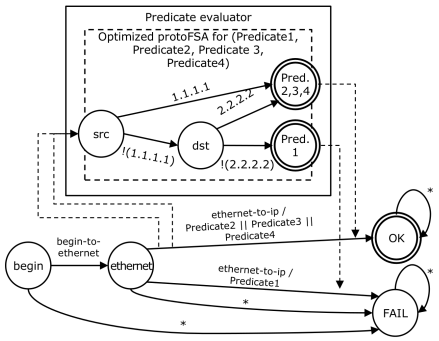


Figure 14. Example of the protoFSA created in Figure 12, composing predicates P_1, P_2, P_3 and P_4, and the resulting optimized protoFSA.

D. About optimality

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

E. Predicates and ranges

The protoFSA creation mechanism presented in Section V-C may lead to an automaton with a huge number of symbols, which may represent a problem when defining the transitions exiting from each state, since their cardinality is equal to the number of symbols. In fact, the explosion in the number of transitions affects both the memory occupancy and the

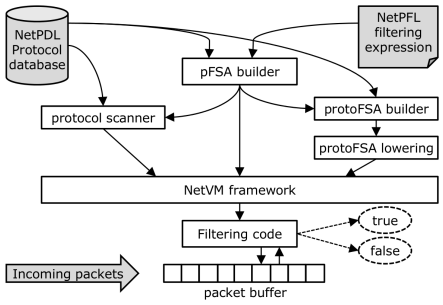


Figure 15. Overview of the building blocks in our prototype.

computational complexity of the FSA algorithms. In order to overcome this problem, whenever possible we group symbols into ranges, using the full enumeration of the symbols only when needed (e.g., when ranges become very complex). For instance, if a predicate specifies a precise IP address such as in Figure 13, we define only two transitions, one for the path that leads to success (associated with the proper IP address, e.g., $\{$`1.1.1.1`$\}$) and the other for all the remaining symbols (e.g., Σ `-` $\{$`1.1.1.1`$\}$).

VI. Implementation

The proposed pFSA model has been implemented in the NetBee library [19], which features an experimental compiler that creates run-time code for the NetVM [17] virtual machine. The front-end compiler [18] takes the filtering expression expressed as a NetPFL [16] string and a NetPDL [15] protocol database to generate an in-memory representation of the pFSA filter. This code is then translated into NetIL code, a NetVM-specific assembly-like language. The generated code can be executed in a NetVM interpreter, or compiled Just-In-Time (JIT) if a backend compiler is available for the target architecture. The pFSA abstraction has been implemented inside the front-end of the aforementioned high-level compiler.

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

A. Overview

Figure 15 shows an overview of the code generation system implemented by our prototype, which mimics the general architecture presented in Figure 6. The *pFSA builder* takes the protocol encapsulation graph (dynamically extracted by the NetPDL protocol database) and the filtering expression and creates the actual pFSA that implements the packet filter. The tokens that allow moving from one pFSA state to another are generated by the *protocol scanner*, which (again) uses the NetPDL protocol database to translate encapsulation rules into running code. Finally, the *protoFSA builder* creates a set of protoFSA, each one dedicated to a single Cartesian product originated by the pFSA. Each protoFSA is then handled by the *protoFSA lowering* module, which takes care of some implementation-dependent optimizations, presented later in Section VI-C. All the aforementioned blocks generate the proper data structures according to the primitives exported by the NetVM framework, which finally merges all the code in order to build the actual filtering program.

B. Protocol scanner

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

C. Predicate evaluator

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

In our implementation both steps are confined into a separate library that takes into account range-based optimizations: all numeric comparisons on a protocol field (involving both range and equality operators) are rearranged into a tree, organized to easily recognize impossible outcomes (e.g., `tcp.dport == 80 and tcp.dport > 1024`). In addition, particular attention has been made in order to lower the code originated by each protoFSA state in the most efficient way. When all transitions exiting from a state include only precise values (such as in the filter `tcp.sport == 80 or tcp.sport == 8080`), the code will be translated into a `switch-case`. When dealing with ranges, instead, the protocol field is initially checked against the bounds of the wider range and, if necessary, against the smaller ones; the comparison for equality against some constants is deferred at the end, if the value is found to lie in the appropriate range. An example can be seen in Figure 16.

D. Code generation

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

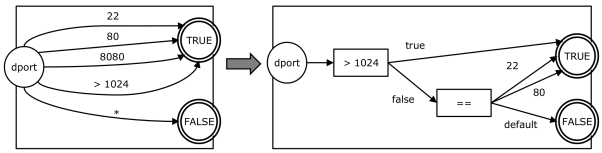


Figure 16. Example of a predicate that specifies multiple comparisons against the same protocol field, generated with the filter: `tcp.dport > 1024 or tcp.dport == 80 or tcp.dport == 22 or tcp.dport == 8080`. Note the tree structure and the removal of the redundant checks.

of checks on protocols and fields. However, from the practical point of view, the characteristics of the NetVM framework allow us to speculate that, if the generated code is not optimal, it is very close to it. For instance, the NetVM framework implements many data-flow and control-flow optimizations (more details in [17]) and our experimental evaluation proves that our speculation is correct in case of the most common filtering expressions, while in other more complex cases the code is rather close to optimality.

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

E. Safety

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

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

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

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

Table I
SAMPLE FILTERS

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

VII. VALIDATION

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

Three different test categories were set up to evaluate different aspects of our solution: *(i)* compile-time performance, *(ii)* run-time performance and *(iii)* scalability. These tests are largely inspired at those in [7] and were performed in a very similar environment. All tests were performed on a workstation equipped with an Intel E8400 Core 2 Duo dual-core processor with 4 GiB of RAM, running a 64-bit version of Ubuntu Linux 10.04. Time measurements were performed either using the RDTSC assembly instruction or, for reasonably longer periods of time, the `gettimeofday()` UNIX function. Memory footprint measurements were performed by using the GNU `time` command or, where applicable, using the Java VM memory management methods. All test processes were bound to a single processor, with hot disk and processor caches, and the machine was otherwise unloaded.

A. Filter compilation time

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

Figure 17 portraits, in logarithmic scale, the time needed for pFSA and SPAF to compile the filters above, either when run with the *core* database or with the *full* one. pFSA running times are broken down in actual *compilation time* (the time needed to get to the final automaton and generate the NetIL code from it) and *optimization time* (the time needed for the data-flow and control-flow optimizations to run over the NetIL code). JIT compilation time for pFSA is not displayed; neither the time required to compile the C code generated by SPAF. SPAF computation times are missing for filters 3 to 5 when executed with the *full* database, because we interrupted those tests when their processing time exceeded 24 hours.

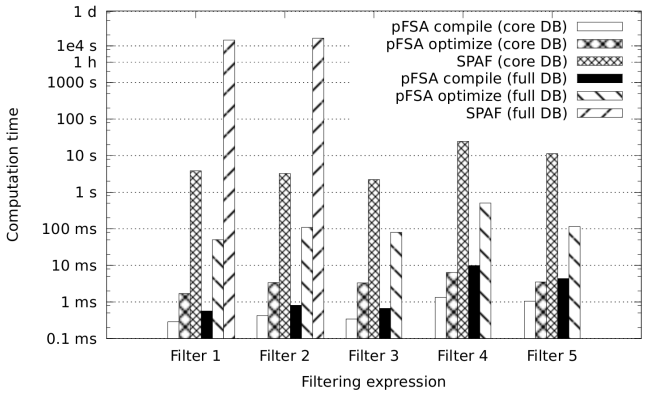


Figure 17. Comparison of the time needed by pFSA and SPAF to compile and optimize a filter.

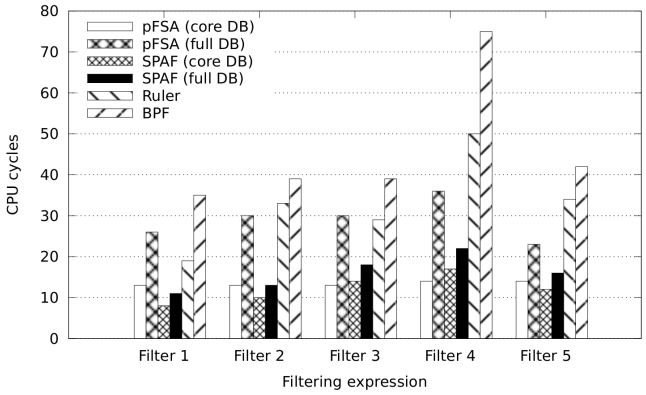


Figure 18. Maximum number of CPU cycles needed to evaluate a packet for each filter.

Table II
MEMORY USAGE AND NUMBER OF STATES

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

Figure 17 shows that the pFSA filter compilation process is several orders of magnitude faster than SPAF, even if we include the optimization time (which is not formally part of the model). The reason can be found in the greater efficiency of the building process of the automaton, which is due to the choice to consider protocols and fields when building the FSA instead of relying on unlabeled bytes in the packet, generating a far smaller number of states. This has a huge impact on the overall building process, as the complexity of FSA manipulation algorithms is usually exponential in the number of states, while other sources of inefficiencies (e.g., SPAF is coded in Java) are less important.

Table II displays, for each filter, the maximum amount of memory required for the compilation process by pFSA and SPAF (which depends on the intermediate transformation of the automaton), and the number of states included in the *final* automaton. These results apply to the *core* database and, in case of pFSA, they include also the count of intermediate protoFSA states. These numbers prove that there is a clear difference between the two implementations: even if the memory usage represents a *peak* measurement, while the number of states is measured at the *end* of the filter compilation, those numbers give a rough indication of the different efficiency of those algorithms.

B. Filter runtime performance

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

Figure 18 shows the *maximum* number of CPU cycles needed to analyze a packet, for each filter, implementation and (if applicable) protocol database. We have chosen to record the maximum number of cycles (instead of the average) to reduce the impact of non-matching and very short packets on the experiment. The best results are achieved by pFSA and SPAF: their performance is roughly the same, especially when using the *core* protocol database. SPAF leverages a more aggressive bounds checking algorithm that represents an advantage when many packet accesses are needed, such as in case of the *full* database. In any case, pFSA results always faster than Ruler and BPF, even with the *full* database.

C. Filter scalability

The last round of experiments checks how pFSA performs when increasing the number of TCP sessions[10] in a given filter. Compilation times for filters with increasing number of sessions are tested first: results are shown in Figure 19.

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

When the *full* protocol database is used, both compilation and optimization times grow exponentially in the number of sessions: for practical reasons, only the first data points are drawn in Figure 19. While unfortunate, this behavior is fully

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

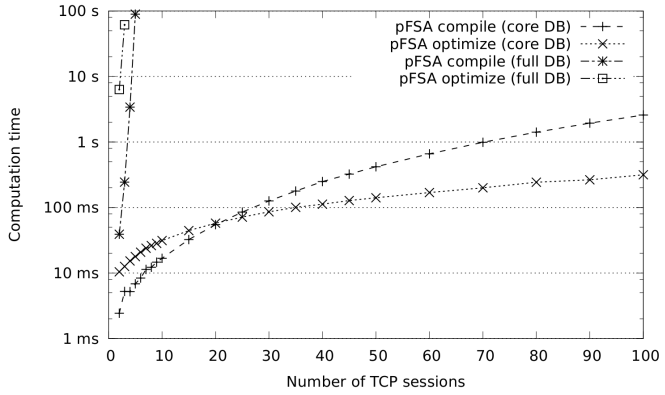


Figure 19. Compile and optimization times needed by pFSA to compile TCP session filters.

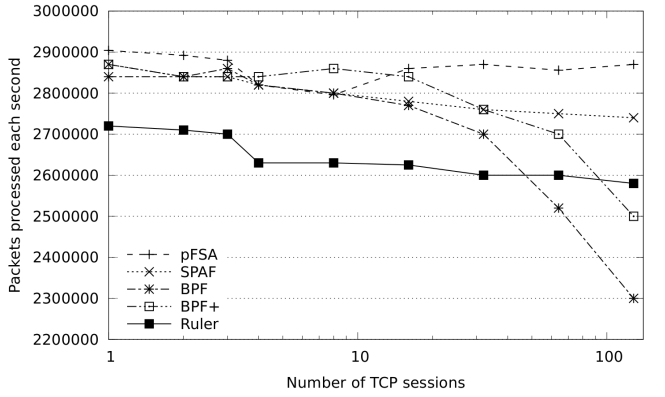


Figure 20. Overall runtime performance w.r.t. TCP session filters.

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

Our last test still focuses on TCP sessions scalability, but instead evaluates the runtime performance of pFSA. We used the same packet trace of Section VII-B and tested the raw packet throughput: one-time computations (e.g., filter compilation) were not considered, but run-time overheads (e.g., per-packet libpcap library calls) are included in the results. Since some of the competing approaches cannot handle multiple

Table III
NUMBER OF TOKENS IN THE FILTERING STRING NEEDED TO FILTER A TUNNELED IPV4 INSTANCE WITH A GIVEN DESTINATION ADDRESS

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

levels of encapsulation, in this test we configured pFSA and SPAF to use the *core* protocol database. Figure 20 shows that pFSA does not suffer any significant runtime performance degradation when the number of filtered sessions increases. This is an expected scenario, because the generated FSA grows wider, but not deeper; as the number of sessions grows, more and more states are added in parallel to the old ones, but the average distance from the starting state to the accepting ones does not change.

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

D. Ease of use

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

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

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

Table III has a rundown of the increasing complexity (in terms of number of tokens in the filtering string) of a BPF

filter, compared to the constant complexity required by pFSA and SPAF.

VIII. Conclusion

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

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

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

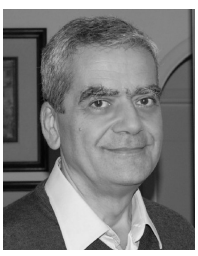
[17] O. Morandi, F. Risso, P. Rolando, S. Valenti, P. Veglia, Creating Portable and Efficient Packet Processing Applications. In *Springer Design Automation for Embedded Systems*, Vol. 15, No. 1, pp. 51-85, March 2011.

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

[19] NetBee library, available at: http://www.nbee.org

Marco Leogrande (marco.leogrande@polito.it) received his M.Sc. Degree in Computer Engineering from Politecnico di Torino in 2009 with a thesis about front-end optimizations for a dynamic packet filter compiler. He is currently a Ph.D. student in Information and System Engineering. His main area of research focuses on performance optimization and feature development in dynamic packet filters.

Fulvio Risso (fulvio.risso@polito.it) is currently assistant professor with the Department of Control and Computer Engineering of Politecnico di Torino, Italy. His current research activities focus on efficient packet processing, traffic analysis, programmable networks.



Luigi Ciminiera passed away on September 3rd, 2012. At the time of his death, prof. Ciminiera was professor of computer engineering with the Department of Control and Computer Engineering of Politecnico di Torino, Italy. His research interests included grids and peer-to-peer networks, distributed software systems, and computer arithmetic. He was coauthor of two international books and more than 100 contributions published in technical journals and conference proceedings.