SPAF: Stateless FSA-Based Packet Filters

# SPAF: Stateless FSA-based Packet Filters

Pierluigi Rolando, Riccardo Sisto, Fulvio Risso

***Abstract*—We propose a stateless packet filtering technique based on Finite-State Automata (FSA). FSAs provide a comprehensive framework with well-defined composition operations that enable the generation of stateless filters from high-level specifications and their compilation into efficient executable code without resorting to various opportunistic optimization algorithms. In contrast with most traditional approaches, memory safety and termination can be enforced with minimal run-time overhead even in cyclic filters, thus enabling full parsing of complex protocols and supporting recursive encapsulation relationships.**

**Experimental evidence shows that this approach is viable and improves the state of the art in terms of filter flexibility, performance and scalability without incurring in the most common FSA deficiencies, such as state space explosion.**

## I. Introduction

PACKET filters are a class of packet manipulation programs used to classify network traffic in accordance to a set of user-provided rules; they are a basic component of many networking applications such as shapers, sniffers, demultiplexers, firewalls and more.

The modern networking scenario imposes many requirements on packet filters, mainly in terms of processing speed (to keep up with network line rates) and resource consumption (to run in constrained environments). Filtering techniques should also support modern protocol formats that often include cyclic or repeated structures (e.g. MPLS label stacks, IPv6 extension headers). Finally it is also crucial that filters preserve the integrity of their execution environment, both in terms of memory access safety and termination enforcement, especially when running as an operating system module or on the bare hardware. Although at first sight this aspect might not seem crucial, it is a fact that many of the limitations built into existing packet filters derive directly from safety issues: as an example, the impossibility of automatically proving termination for a generic computer program led the BPF [1] designers to generate acyclic filters only, thus preventing the parsing of packets with multiple levels of encapsulation or repeated field sequences.

Existing packet filters focus invariably on subsets of these issues but, to the best of our knowledge, do not solve all of them at the same time: as an example, two widely known generators, BPF+ [2] and PathFinder [3], do not support recursive encapsulation; NetVM-based filters [4], on the other hand, have no provision for enforcing termination, either in filtering code or in the underlying virtual machine.

This paper presents SPAF (Stateless PAcket Filter), a FSA-based technique to generate fast and safe packet filters that are also flexible enough to fully support most layer 2 to layer 4 protocols, including optional and variable headers and recursive encapsulation. The proposed technique specifically targets the lower layers of the protocol stack and does not directly apply for deep packet inspection nor for stateful filtering in general. Moreover, for the purpose of this paper we consider only static situations where on-the-fly rule set updates are not required. While these limitations exclude some interesting use cases, SPAF filters are nevertheless useful for a large class of applications, such as monitoring and traffic trace filtering, and can serve as the initial stage for more complex tools such as intrusion detection systems and firewalls.

A stateless packet filter can be expressed as a set of predicates on packet fields, joined by boolean operators; often these predicates are not completely independent from one another and the evaluation of the whole set can be short-circuited. One of the most important questions in designing generators for high-performance filters is therefore how to efficiently organize the predicate set to reduce the amount of processing required to come to a match/mismatch decision. By considering packet filtering as a regular language recognition problem and exploiting the related mathematical framework to express and organize predicates as finite-state automata, SPAF achieves by construction a reduction of the amount of redundancy along any execution path in the resulting program: any packet field is examined at most once. This property emerges from the model and it always holds even in cases that are hard to treat with conventional techniques, such as large-scale boolean composition. Moreover, thanks to their simple and regular structure, finite automata also double as an internal representation directly translatable into an optimized executable form without requiring a full-blown compiler. Finally, safety (both in terms of termination and memory access integrity) can be enforced with very low run-time overhead.

The rest of this paper is structured as follows: Section II presents an overview of the main related filtering approaches developed to this date. Section III provides a brief introduction to the FSAs used for filter representation and describes the filter construction procedure. Section IV focuses on executable code generation and on enforcing the formal properties of interest, while Section V presents the experimental evidence collected to evaluate the new approach and to support our claims. Finally, Section VI reports conclusions and also highlights possible future developments.

## II. Related works

Given their wide adoption and relatively long history, there is a large corpus of literature on packet filters. A first class of filters is based on the CFG paradigm; the best-known and

The authors are with Politecnico di Torino, Dipartimento di Automatica e Informatica, e-mail: {pierluigi.rolando, riccardo.sisto, fulvio.risso}@polito.it

This work has been partially supported by The Cisco University Research Program Fund, a corporate advised fund of Silicon Valley Community Foundation.

most widely employed one is probably BPF [1], the Berkeley Packet Filter. BPF filters are created from protocol descriptions hardcoded in the generator and are translated into a bytecode listing for a simple, ad-hoc virtual machine. The bytecode was originally interpreted, leading to a considerable run-time overhead impact which can be reduced by employing JIT techniques [5]. BPF disallows backward jumps in filters in order to ensure termination, thus forgoing support for e.g. IPv6 extension headers; memory protection is enforced by checking each access at run-time. Multiple filter statements can be composed together by boolean operators but in the original BPF implementation only a small number of optimizations are performed over predicates, leading to run-time inefficiencies when dependent or repeated predicates are evaluated. Two relevant BPF extensions are BPF+ and xPF. BPF+ [2] adds local and global data-flow optimization algorithms that try to remove redundant operations by altering the CFG structure. xPF [6] relaxes control flow restrictions by allowing backward jumps in the filter CFG; termination is enforced by limiting the maximum number of executed instructions through a run-time watchdog built into the interpreter but its overhead was not measured and extending this approach to just-in-time code emission has not been proposed and might prove difficult.

A further CFG-based approach, unrelated to BPF, is described in [4]. Its main contribution is decoupling the protocol database from the filter generator by employing an XML-based protocol description language, NetPDL [7]. Filtering code is executed on the NetVM [8], a special-purpose virtual machine targeting network applications that also provides an optimizing JIT compiler that works both on filter structure and low-level code. The introduction of a high-level description language reportedly does not cause any performance penalties; this approach, however, delegates all safety considerations to the VM and does not provide an effective way to compose multiple filters.

In general CFG-based generators benefit from their flexible structure that does not impose any significant restriction on predicate evaluation order; for the same reason, however, they are prone to the introduction of hard-to-detect redundancies, leading to multiple unnecessary evaluations if no further precautions are taken. Even when optimizers are employed and are experimentally shown to be useful, they work on an opportunistic basis and seldom provide any hard guarantees on the resulting code.

A second group of filter generators chooses tree-like structures to organize predicates. PathFinder [3] transforms predicates into template masks (atoms), ordered into decision trees. Atoms are then matched through a linear packet scan until a result is reached. Decision trees enable an optimization based on merging prefixes that are shared across multiple filters. PathFinder is shown to work well both in software and hardware implementations, but it does not take protocol database decoupling into consideration and no solution to memory safety issues is proposed for the software implementation. FSA-based filters share a degree of similarity with PathFinder as packets are also scanned linearly from the beginning to the end but predicate organization, filter composition and safety considerations are handled differently. DPF [9] improves over PathFinder by generating machine code just-in-time and adding low-level optimizations such as a flexible `switch` emission strategy. Moreover, DPF is capable of aggregating bounds checks at the atom level by checking the availability of the highest memory offset to be read instead of considering each memory access in isolation; our technique, described in Section IV-E, acts similarly but considers the filter as a whole, thus further reducing run-time overhead.

While organizing predicates into regular structures makes it easier to spot redundancies and other sources of overhead, it also introduces different limitations: as an example, generators restricted to the aforementioned acyclic structures do not fully support tunneling or repeated protocol portions. Moreover, it has been noted that performing prefix coalescing is not sufficient to catch certain common patterns, resulting in redundant predicate evaluation [2].

A third approach is to consider packet filtering as a language recognition problem. Jayaram et al. [10] use a pushdown automaton to perform packet demultiplexing; filters are expressed as LALR(1) grammars and can be therefore effectively composed using the appropriate rules. This solution improves filter scalability but there are downsides related to the pushdown automaton: a number of specific optimizations are required to achieve good performance. It is also quite unwieldy to express protocols and filter rules as formal grammars that must be kept strictly unambiguous: the authors marginally note that the simpler FSA model would be sufficient for the same task.

The adoption of finite state automata in packet processing is not new, as finite automata are already used to perform deep payload inspection especially in intrusion detection systems (IDS) [11] [12] [13] [14] [15] [16]. While the general approach is similar, the practical issues involved in deep payload inspection can be rather different from those encountered when employing FSAs for layer 2 to layer 4 filtering. As an example, payload regular expressions tend to be free-form and often present patterns that negatively affect the size of the automata resulting from composition, causing a so-called state space explosion [16] [17]. This issue is arguably the limiting factor to FSA adoption for payload inspection and it certainly deserves investigation; however, our scenario might be different enough (protocol headers often have a much more rigid structure) so that its impact is mitigated. Among many FSA-based packet processors the nearest match for SPAF is probably Ruler [18], a packet rewriting system designed for anonymizing traffic traces that can also be used for packet filtering. It is based around an NFA extension that supports rewriting and its engine is specifically tailored to Intel IXP network processors. Being automata-based, Ruler shares a degree of similarity with SPAF but its design goals are sufficiently different to produce noticeably different final results. Ruler's design aims at supporting large free form regular expression sets and the modifications required for rewriting; less attention has been paid to aspects that are necessary for achieving good layer-2 to layer-4 filtering performance, thus trailing SPAF in experimental comparisons (Section V). Safety issues are also not taken into account conveniently (memory bounds are checked at each access) and its source language is not general enough to specify complex filter statements or certain commonly encountered protocol structures, such as the IPv6 extension headers: while it is possible to resort to regular expressions that represent the entire packet structure, this formulation presents an avoidable layer of complexity for the final user.

Apart from the specialized solutions for fast packet filtering mentioned above, one of the most widely used packet filtering programs is the NetFilter framework[1]. NetFilter is a component of the Linux kernel that performs packet filtering, firewalling, mangling operations (e.g. network address translation) and more, acting through a set of hooks and callbacks that intercept packets as they traverse the networking stack. In contrast with all the aforementioned approaches, NetFilter uses the relatively simple method of applying all the specified rules in sequence when performing packet filtering, leading to poor performance and scalability; moreover it appears not possible to specify an arbitrary predicate, filters being limited to preset protocols and statements that are specialized by specifying actual network addresses and ports.

Besides the generation technique, there have also been improvements along other dimensions such as architectural considerations, as demonstrated by xPF, FFPF [19] and nCap [20], or dynamic rule sets support, as shown by the SWIFT tool [21]. We consider these aspects out of scope for the purpose of this paper, being either orthogonal to the technique we present or object of future works.

## III. Filter generation technique

The purpose of a stateless packet filter generator is to create a program that, given a finite-length byte sequence (a packet) as its input, returns a binary match/mismatch decision. The input of the generator itself consists of a set of filter rules provided by the user that specify the desired properties of matching packets; each rule, in turn, consists of multiple predicates expressed in a simple high-level language (where header fields and protocols appear symbolically), combined together with boolean operators. In older generators the set of supported protocols was fixed; in modern ones protocol header formats are kept into an external database that can be updated without modifying the generator.

In order to develop a successful FSA-based filtering technique it is first of all necessary to show that any filter of interest can be expressed as a finite automaton, then provide a method to transform a high-level filter statement and a protocol database into FSA form; finally, the resulting automaton must be translated into an efficiently executable form.

### *A. Finite-State Automata*

A Finite-State Automaton (FSA) is a quintuple $(A, S, s_0, t, F)$, where $A$ is an alphabet of input symbols, $S$ is the set of states, $s_0 \in A$ an initial state, $t \subseteq S \times A \times S$ the transition relation and $F \subseteq S$ the set of accepting states.

FSAs can be used to formally define regular sets of arbitrarily long symbol strings; given a finite automaton and a string, it is easy to decide whether the string belongs to the set represented by the FSA or not. The parallelism with stateless packet filters is immediate: each packet can be regarded as a string of bytes and a filter statement defines a set of packets that must be recognized. In order to adapt the FSA model to our purposes, it is sufficient to define $A$ as the set of all the possible 8-bit strings plus the symbol $\epsilon$, used to label transitions that can be taken without consuming any input.

The only remaining requirement is to show that any interesting set of packets is regular; this is immediately proved by noting that any finite set is regular and that there are only finitely many packets because they are limited in length by technological considerations[2]. It is therefore theoretically possible, for any conceivable stateless packet filter, to build a corresponding FSA. Even recursive structures can be supported: while in the general case a more powerful formalism (such as a push-down automaton) is required, restricting the scope of application to finite sets makes regular automata sufficient.

### *B. Protocol database compilation*

The first phase in the SPAF generation process consists of parsing the protocol database and building template automata that recognize all the correctly-formatted headers for a given protocol. These automata will be reused and specialized in later phases to create the final filter.

In order to decouple filter generation from the protocol database, we have employed an XML-based protocol description language (NetPDL [7]) designed to describe the on-the-wire structures of network protocols and their encapsulation relationships. NetPDL descriptions are stored in external files that can be freely edited without modifying the generator itself.

A precise description of NetPDL is beyond the scope of this paper; nevertheless we shall provide a quick overview of the features supported by the FSA generator. The language provides a large number of primitives that enable the description of header formats of layer 2 to 7 protocols, but for the scope of this work we have restricted our support to those designed for layer 2 to 4 decoding. The basic building block of a protocol format is the header field, a sequence of bytes or bits that can be either fixed or variable in size. Adjacent fields are by default laid out in sequence but more complex structures such as optional or repeated sections can be created using conditional choices and loops; these statements are controlled by expressions that can contain references to the values of previously-encountered fields.

A second NetPDL portion contains a sequence of control flow operations (`if`, `switch`) that predicate encapsulation relationships: in general, the control flow is followed until a `nextproto` tag is encountered, specifying which is the next protocol to be found in the packet. A NetPDL database thus describes an oriented encapsulation graph where the vertexes are protocols and the edges are encapsulation relationships.

[1] NetFilter is available at http://www.netfilter.org/

[2] This is not true in general because FSAs can be used to filter unlimited-length symbol strings as those encountered e.g. in payload inspection with transport session reconstruction. Given the scope of this paper (layer 2 to 4 packet filtering) this is not a concern.

```
<protocol name="ipv6">
  <format>
    <fields>
      <field type="bit" name="ver" mask="0xF0000000" size="4"/>
      <field type="bit" name="tos" mask="0x0F000000" size="4"/>
      <field type="bit" name="flabel"mask="0x00FFFFFF" size="4"/>
      <field type="fixed" name="plen" size="2"/>
      <field type="fixed" name="nexthdr" size="1"/>
      <field type="fixed" name="hop" size="1"/>
      <field type="fixed" name="src" size="16"/>
      <field type="fixed" name="dst" size="16"/>

      <loop type="while" expr="1">
        <switch expr="nexthdr">
          <case value="0"> <includeblk name="HBH"/> </case>
          <case value="51"> <includeblk name="AH"/> </case>
          ...
          <default>
            <loopctrl type="break"/>
          </default>
        </switch>
      </loop>
    </fields>
  </format>

  <encapsulation>
    <switch expr="nexthdr">
      <case value="4"> <nextproto proto="#ip"/> </case>
      <case value="6"> <nextproto proto="#tcp"/> </case>
      <case value="17"> <nextproto proto="#udp"/> </case>
      ...
    </switch>
  </encapsulation>
</protocol>
```

Fig. 1. IPv6 NetPDL excerpt

Currently the graph begins with a single, user-specified root that usually represents the link layer protocol but an extension to multiple ones would be trivial. Starting from this root, the FSA generator follows the encapsulation graph and builds a FSA for every reachable protocol using the method explained later in this section.

As an example, a simplified NetPDL description of the IPv6 header format is presented in Fig. 1. IPv6 starts with a sequence of fixed-size fields; bitfields (such as `ver`) are specified by the `mask` attribute. The initial portion is followed by a set of extension headers, each one containing a "next header" information (`nexthdr`). This sequence is of unspecified (but implicitly finite, as any packet is finite) length and it is described using a `switch` nested within a `loop`: at each iteration the newly-read `nexthdr` field is evaluated and, if no more extension headers are present, the loop terminates. Encapsulation relationships are also specified in a similar fashion, by jumping to the correct protocol depending on the value of the last `nexthdr` encountered.

SPAF currently supports the full versions of the most common layer 2-4 protocols in use nowadays, such as Ethernet, MPLS, VLAN, PPPoE, ARP, IPv4, IPv6, TCP, UDP and ICMP; this set can be easily extended as long as no stateful capabilities are required.

An important point regarding FSA creation from NetPDL descriptions is that, as long as it is correctly performed, it is not a critical task for filter performance: any resulting automaton ultimately will be determinized and minimised, yielding a canonical representation of the filter that does not depend on the generation procedure. For this reason, and given the complexity involved, the NetPDL to FSA conversion procedure is not fully described in this paper and it can be regarded as an implementation detail. Nevertheless, in order to exemplify how the conversion can be done, we report the key steps for translating the NetPDL snippets of Fig. 2 into the corresponding automata.

The purpose of this initial conversion step is not to generate automata immediately suitable for filtering; on the contrary, the results are templates for the following generation steps, representing the "vanilla" version of protocol headers, with no other conditions imposed, to be specialized according to the filter rules. Since they are strictly related to header format, any input-consuming transition in these templates can be related to a specific portion of one[3] header field; this information must be preserved to accommodate the imposition of filtering rules. For this reason template automata are augmented by marking all the relevant transitions with the related field's name[4].

The simplest example is generating an automaton that parses a fixed-length header field (Fig. 2a): it is sufficient to build a FSA that skips an appropriate amount of bytes, resulting in Fig. 2b. During the construction process header fields are given well-defined start and end[5] states that are used as stitching points to join with any predecessors or successors by $\epsilon$-transitions, as required.

A more complex example involving a conditional choice is shown in Fig. 2c. The generation procedure starts by creating automata representations for all the initial fields in the NetPDL description; upon encountering the `switch` construct, however, the generator backtracks the transition graph until it encounters the `type` field. Once found, all the states/transitions that follow `type` (the `A` block in the figure) are replicated. The original copy is left as-is while in the replica the transitions for `type` are specialized to recognize the bytes of interest for the `switch`, so the right path will be taken depending on the actual input values. Finally the correct trailing block (`B` or `C`) is joined in the right place.

The last example (Fig. 2e and Fig. 2f) shows the automata generated for a header structure similar to the IPv6 extension headers case. In this case a loop is interlocked with a `switch` construct and a greater amount of block replication is required to ensure that independent paths exist into the automaton for every possible combination of the current `nexth` value (upon which the outcome of the `switch` depends) and the next `nexth` value, that might cause the `loop` to end.

Encapsulation relationships are handled in a similar fashion, by spawning new paths in the automaton graph that end with a special state marked with the protocol that should follow. The exact usage of these marked states is explained in Section III-D.

The generation procedure acts to counter the absence of explicit storage locations in the FSA model; when it becomes necessary to use the values of previously encountered fields for subsequent computations the only solution is to spawn a number of parallel branches within the automaton, each one associated with a specific value of the field under consideration.

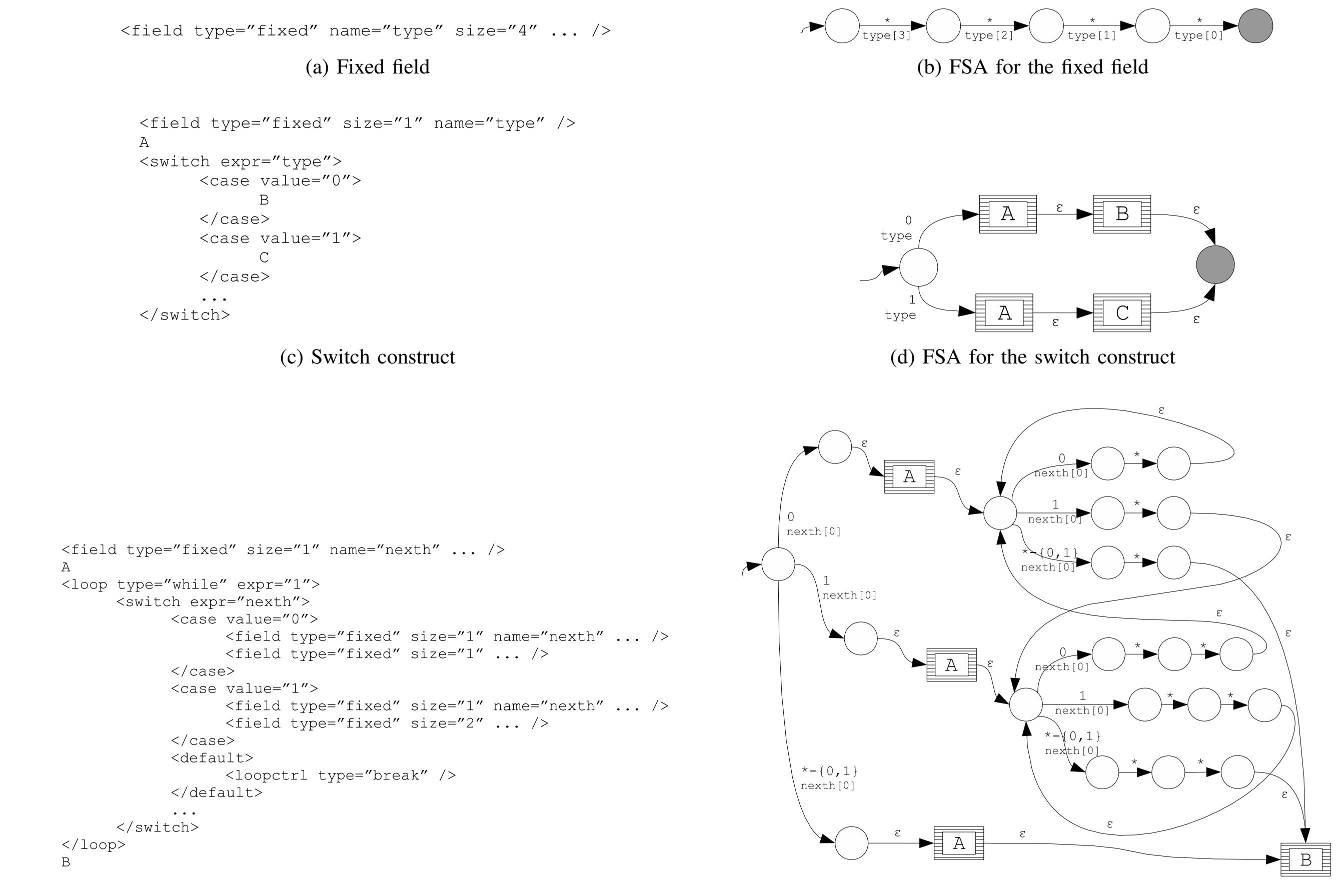
### *C. Filter rule imposition*

The second generation step takes filtering rules into consideration. The user provides the filter generator with an arbitrarily complex rule that is split into a parse tree where leaves are

[3] Or possibly more, in case of bitfields.

[4] These names appear in figures only when relevant.

[5] Not necessarily final.



```
<field type=”fixed” name=”type” size=”4” ... />
```

(a) Fixed field

(b) FSA for the fixed field

```
<field type=”fixed” size=”1” name=”type” />
A
<switch expr=”type”>
      <case value=”0”>
            B
      </case>
      <case value=”1”>
            C
      </case>
      ...
</switch>
```

(c) Switch construct

(d) FSA for the switch construct

```
<field type=”fixed” size=”1” name=”nexth” ... />
A
<loop type=”while” expr=”1”>
      <switch expr=”nexth”>
            <case value=”0”>
                  <field type=”fixed” size=”1” name=”nexth” ... />
                  <field type=”fixed” size=”1” ... />
            </case>
            <case value=”1”>
                  <field type=”fixed” size=”1” name=”nexth” ... />
                  <field type=”fixed” size=”2” ... />
            </case>
            <default>
                  <loopctrl type=”break” />
            </default>
            ...
      </switch>
</loop>
B
```

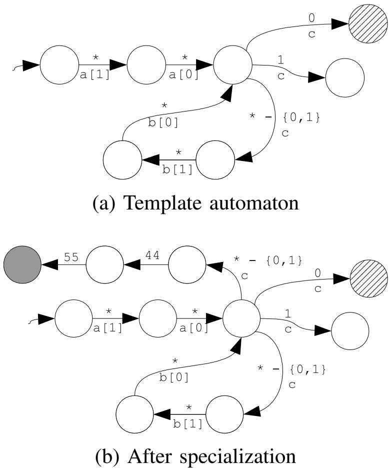
(e) Interlocked loop and switch

(f) FSA for interlocked loop and switch

Fig. 2.   Filter generation examples

predicates (e.g. ip.src = 10.0.0.1) and internal nodes are boolean operators. Working on a single predicate at a time, the generator creates a copy of the related protocol template automaton and specializes it so that the resulting FSA reaches a success state if and only if input data makes the predicate true. In order to perform this step the generator uses template annotations to find the transitions corresponding to the field specified by the predicate, then creates a parallel path labeled with the expected values. The last state of this specialized transition chain is marked as final to implement the required behavior.

An example of the specialization procedure is reported in Fig. 3. Let us consider a simple protocol with 3 header fields (a, b and c) and a structure leading to the template automaton of Fig. 3a. Encapsulation conditions state that if c is 0 then the protocol encapsulates itself as its payload; this translated into marking the dashed state in the figure with the appropriate identifier. Finally, let us assume the user specifies a single predicate: b = AA BB. When parsing this predicate the generator creates a replica of the graph in Fig. 3a, then proceeds to locate the b field. During the specialization step a chain of 2 transitions matching the 44 55 byte sequence and ending with a final state is generated and then pasted in parallel to the branch leading to field b. The result is shown in Fig. 3b where it appears clear that the only path to success goes through the specialized version of the b field, as required.



(a) Template automaton

(b) After specialization

Fig. 3.   Automaton specialization

The generator does not directly impose the predicate over the original field but instead creates a parallel state chain: this is because, as in the example above, multiple instances of the field under consideration might be present in the actual header and overwriting the original transitions would cause the predicate to be matched only against its first occurrence. The resulting automaton can be non-deterministic, but this poses no issues to the generator. A simpler case is when a predicate

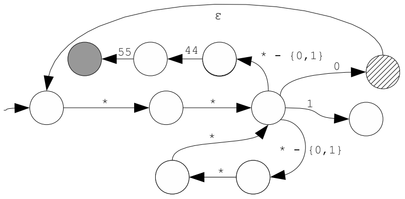


Fig. 4. Encapsulation handling

only requires that the packet includes a correctly formatted header of a given protocol. In this case, the specialized FSA is a copy of the template automaton for that protocol where all marked states are made final.

During this step it is possible for the user-specified rules to involve fields that are already specialized due to NetPDL constraints; as an example, NetPDL forces the IPv4 header length field to be at least 5 but the user could specify a rule that requires it to be lesser than that. No special precautions are taken to handle these situations because the mismatch will be automatically solved later through composition, determinization and minimisation. In the case of the example, there will be two and-ed specialized automata, one predicating the packet includes a correct IPv4 header and the other one predicating the length field is less than 5. If incompatible conditions are predicated then the composite automaton will degenerate to a single, non-accepting state, meaning that the recognized set of packets is empty.

### D. Filter composition

Once the generator has prepared a specialized FSA for each rule predicate, what remains to be done is to assemble all these automata together to create the final filter.

The composition step performs 2 different operations:

1) joining multiple automata related to different protocols following encapsulation rules;
2) combining multiple automata by boolean operators.

Encapsulation is handled one predicate at a time, by introducing $\epsilon$-transitions that bridge automata related to different protocols, following the markings from the first generation phase. As an example, in the case described in Fig. 3, the automaton that results after considering encapsulation is shown in Fig. 4: the marked state has an additional outgoing $\epsilon$-transition leading back to the initial state; in this way the predicate will be matched even if the `AA BB` value does not occur in the first encountered header but it occurs in one of the other nested headers.

The second operation is performed in the order dictated by the filter statement parse tree and uses well-known algorithms [22] to implement boolean operations and to make the resulting automaton both deterministic and minimal.

Up to this step, the FSAs used in the generator can be indifferently deterministic (DFA) or non-deterministic (NFA). Here a determinisation step is required to support the implementation of the complement operator, since there are no known algorithms to perform complementation directly on NFAs. The result of the compilation phase is, therefore, a single minimal DFA that represents the filter.

### E. On the properties of FSA-based filters

The FSA model provides by construction a set of properties that can be successfully exploited for our purposes. First of all, the final minimised DFA is the canonical, unambiguous, machine-agnostic representation of a given set of rules on a given set or protocols: FSAs are a formalism adequate to be used as a semantic model for stateless packet filters, fully describing protocol structure and filter rules in a comprehensive representation.

SPAF filters also enjoy practical advantages. Finite automata scan their input strings strictly sequentially so it is true by construction that SPAF examines each packet field at most once, no matter how complex the protocol database or the filtering rules are; this property is also carried across any filter composition operation, which hardly can be achieved with CFG-based approaches. The FSA-based approach is also insensitive to any specific predicate order or other peculiarities of the filter rules: the final result is dependent on the semantics of the filter statement only, regardless of its syntactic form.

As a result of the considerations above, the final automaton can be used as a guideline for a software or hardware implementation by closely mimicking its logical behavior, thus justifying our usage of FSAs as an intermediate representation that decouples generation process from code emission routines and allows in line of principle multi-platform code emission while fully preserving its semantics.

It is worth to note that the built-in efficiency provided by the FSA model is offset by certain limitations. A first point is that SPAF filters examine packet fields in the same order as they appear on the wire. Since the current code emission technique closely mimics the model's behavior, it is possible that some fields are examined before it becomes essential to know their value: the comparison may turn out to be useless on the execution path that is eventually taken. This constitutes a form of partial redundancy that can be handled by many algorithms described in the literature [23] [2] [24]. Respecting a fixed predicate evaluation order also causes suboptimal state and transition counts for the final filter because of the amount of duplication required. In spite of these considerations, experimental evidence (presented in Section V) shows that good quality code can be generated even if this issue is ignored, as the number of partially redundant checks is low when considering real-world protocols and rule sets. Finally, in-order processing of packet bytes is a property shared with many other packet filter approaches[6], where it is not generally regarded as a limitation.

## IV. Executable code generation

The last generation phase is code emission, needed to translate the filter DFA into an executable form. In a parallel to traditional compiler architectures, during this phase the DFA is used as a kind of intermediate representation passed from the front-end (the DFA builder) to the back-end (the code emitter).

While translating a DFA into an executable form is by no means a difficult or innovative task, it is nevertheless critical

[6] In fact it might be argued that most packet processing applications read their input in-order.

for our objectives and in particular both for performance and safety. In line of principle the filter structure could be translated into a regular expression and then fed to a general-purpose matching engine such as the one provided by Flex[7], the PCRE library[8] or Ruler. In practice, however, our scope and specific requirements are sufficiently different from mainstream applications of regular expressions to justify the development of an ad-hoc engine: this is because most regex engines implement additional features such as backtracking [25] or rewriting (in the case of Ruler) that, while useful in a general-purpose tool, are an avoidable source of overhead for our purpose.

A further difference from traditional implementations is safety enforcement. A DFA stops when the input symbol sequence is over; this behavior should be emulated in software by an implementation that receives a memory buffer (not a data stream) as its input. While performing a termination check at each step is certainly possible, it is likely to be very expensive as well. Besides termination, replacing the input stream with a memory area also brings a safety problem because of potentially out-of-bounds read operations. Again, these must be avoided with the least possible run-time overhead. While these aspects are usually overlooked, most of the additional filtering capabilities that distinguish FSA-based filters from more traditional approaches (such as loop handling capability) depend on a robust and efficient safety enforcement strategy.

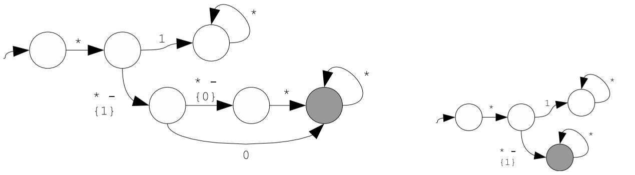
The rest of this section describes the transformations performed over the filter DFA in order to achieve high performance while enforcing safe memory accesses and termination. Afterwards, the code generation technique used in the current back-end is documented.

### *A. Succeed-early algorithm*

Often a user is interested in matching packets against custom rules without necessarily checking for full protocol conformance. In some of these cases it is possible to improve filter performance by disregarding some additional aspects related to protocol structure. As an example, a user may wish to filter packets based on their IP source address only, and get a match as soon as the IP source address field is encountered, even if a malformed TCP header follows; on the contrary the natural behavior of SPAF would be to traverse and validate all known protocol headers.

In order to handle these cases it is possible to run the *succeed-early* algorithm which simplifies a DFA by enumerating all the states that are post-dominated by a final one, marks them as final too, and finally minimizes the automaton. This operation removes any sequence of states that, given enough input symbols, would lead to a final state, regardless of the actual symbols' values.

The effects of the succeed-early algorithm can be seen in Fig. 5: since any sufficiently long path on the lower branch of the DFA graph leads to a final state, the whole branch is marked as final and collapses when minimised.



(a) Original automaton

(b) After succeed-early

Fig. 5. Succeed-early algorithm

In the general case, the succeed-early algorithm modifies the filter's recognized packet set by accepting certain kinds of packets that are truncated or have an otherwise malformed tail; if this is not desirable, the algorithm can be disabled with an apposite filtering predicate that forces full parsing.

### *B. Transition compaction algorithms*

In most filters a large amount of input bytes are never used because neither the protocol database nor the filtering rules predicate anything about them: reading these bytes from memory is an obvious waste of processor cycles and memory bandwidth. In order to improve performance, the generator searches DFA graphs for chains of byte-skipping transitions and, whenever possible, compacts each of them into single multi-byte transitions of the correct length with all the intermediate states removed: this constitutes the *star compaction* algorithm.

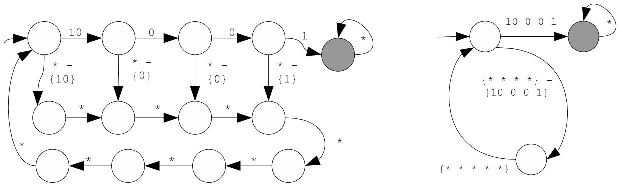
A similar optimization can be performed on non-star transitions to address a different problem: filter DFAs work natively on 8-bit symbols but most modern CPUs are more efficient at processing multi-byte words. The *transition compaction* algorithm takes care of this mismatch by merging multiple subsequent transitions whenever possible, thus allowing the resulting program to operate on larger word sizes. Transition compaction is performed similarly to star compaction: starting from a single state, the transition graph is explored to build long chains of candidate transitions that will be replaced with a multi-byte one. In contrast to star compaction, however, the maximum number of transitions to be compacted is limited to the machine word size. This algorithm trades off states for transitions. Note that the number of transitions might in principle increase exponentially: in order to avoid this issue compaction is not performed on a given state subset if the number of transitions to be introduced is larger than a tunable threshold.

Neither star compaction nor transition compaction remove any path in the FSA graph, so the set of packets recognized by the filter is left unmodified. As an example, we have reported a sample automaton both before (Fig. 6a) and after (Fig. 6b) its star and transition compaction.

The final effect of transition merging is somewhat similar to DFA multi-striding [26] with an important difference: multi-striding keeps all transitions of the same length, so the resulting object is still an automaton with a different input alphabet; on the contrary, transition merging creates

[7] Available at http://flex.sourceforge.net/

[8] Available at http://www.pcre.org/



(a) Original automaton

(b) After compaction

Fig. 6. Compaction algorithms

transitions of different lengths[9] and the result cannot be strictly regarded as an automaton, as there is no longer a well-defined input alphabet. This creates no issues as no further automaton operations are performed after the compaction step in the compilation process; allowing differently-sized transitions provides additional flexibility because only the relevant portions of the transition graph are compressed, without affecting the whole structure.

Transition compaction has also a side effect very similar to an optimization that is also performed by other filtering techniques, sometimes called atom coalescing [9], which works by merging multiple short physically adjacent fields into fewer larger ones, disregarding field boundaries. Since no trace of distinct fields remains at this level in the compilation process, transition compaction automatically exploits every chance of merging atoms.

In addition to reducing the number of operations to perform at run-time and decreasing the amount of data to fetch from packet memory, the post-processed automaton is significantly smaller because many states can be safely eliminated.

### C. C code generation

For simplicity reasons the currently implemented back-end translates compacted DFAs into C functions; a simple JIT compiler for the direct emission of assembly code for any machine would not be difficult to build, if needed.

Given the relatively simple structure and behavior of DFAs, there are multiple possible software implementations. A straightforward automaton implementation can be built by walking through a memory-stored transition table; this solution might incur in overhead due to hard-to-predict memory access patterns. For this reason, we follow the other classical approach where a uniquely identified code fragment is emitted for each state: in this way we get a better exploitation of the CPU prefetch and branch prediction units.

Considering that the input stream is replaced by the aforementioned packet buffer, it is necessary, in the general case, to perform the following steps for each traversed state:

- read the required amount of input bytes from memory;
- appropriately increment the memory offset pointer;
- perform a multi-way conditional comparison by using a `switch` instruction. The cases derive from the outgoing transition labels;
- jump to the destination state.

States with an outgoing byte-skipping transition can be simplified as it is only required to increment the offset pointer by a fixed amount and jump to the next state. A code emission sample for both the complete and simplified cases is depicted in Fig. 7.

The aforementioned code emission strategy ensures no other operations are required at run-time, and all byte-swapping and arithmetic operations are performed at compile-time. The execution environment can be kept minimal as it must provide the input packet buffer and its length only. No other facilities (e.g. memory protection or external libraries) are needed.

A possible optimization is to employ arithmetic operations to reduce the cardinality of multi-way statements. As an example, the check on the IP header length is currently translated into an 11-way conditional construct with 16 labels in each case[10]; the same operation could be replaced with a more traditional masking of the lower 4 bits of the field.

The C code resulting from automata emission is not prone to further optimizations as it is already compact and redundancy-free. In particular, the outcome of each comparison performed cannot be deduced by previous computation and is relevant for the final result; no unnecessary or repeated memory reads are ever performed. Since no arithmetic operations are performed, apart from incrementing the offset pointer by a constant value, the impact of any related optimization performed by the C compiler is expected to be very small. Similarly, loops cannot be unrolled or modified because every guard condition depends on packet data unavailable at compile time. These expectations have been confirmed by visually inspecting the resulting code. The most relevant tasks left to the compiler are low-level machine adaptation procedures such as register allocation and move coalescing [27] and choosing a good emission strategy for the switch instruction [28], which is very common in FSA-based filters but not natively implemented by most CPUs.

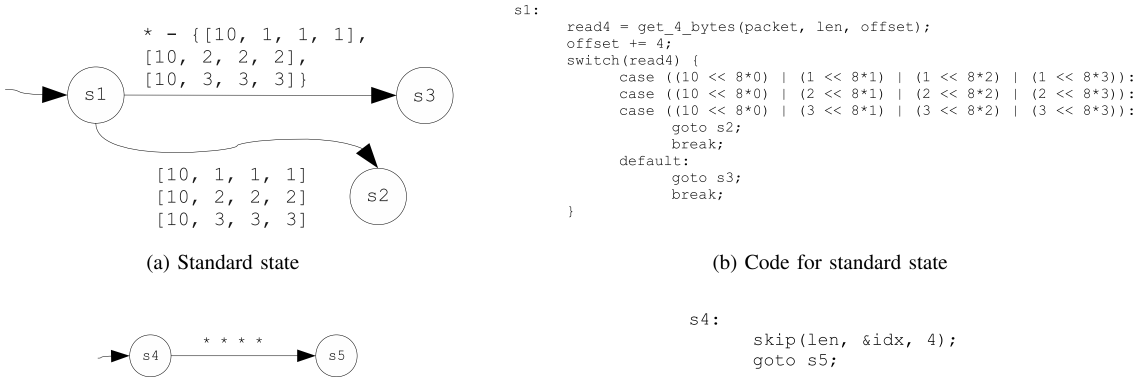
### D. Asymptotic complexity and scalability of FSA-based filters

A major concern in packet filters is their behavior with increasing complexity of the filtering rule set. While many common approaches scale linearly in the worst case, better scalability is desired to adequately support the size of modern rule sets.

If all the required operations (read from memory, offset pointer increment, multi-way conditional choice, unconditional jumps) were executed in constant time, it would be possible to deduce that the worst-case execution time of any FSA-based filter is asymptotically proportional to the length of the input packet. Since packet size is upper bounded by a constant (depending on the actual physical layer), FSA-based filters would run in O(1) time w.r.t. the number of filter rules, independently from their complexity or from the size of the protocol database. While processing speed measurements would still be required to evaluate performance (constant and multiplicative factors might still be large), this asymptotic

[9] Nevertheless, all transitions out of the same state are kept of the same length.

[10] These figures derive directly from the IPv4 protocol definition: header length is a count of 32-bit words, stored as the lower 4 bits of a byte. Some values are invalid, as the minimum IP header length is 20 bytes.



```
s1:
      read4 = get_4_bytes(packet, len, offset);
      offset += 4;
      switch(read4) {
            case ((10 << 8*0) | (1 << 8*1) | (1 << 8*2) | (1 << 8*3)):
            case ((10 << 8*0) | (2 << 8*1) | (2 << 8*2) | (2 << 8*3)):
            case ((10 << 8*0) | (3 << 8*1) | (3 << 8*2) | (3 << 8*3)):
                  goto s2;
                  break;
            default:
                  goto s3;
                  break;
      }
```

```
s4:
      skip(len, &idx, 4);
      goto s5;
```

(a) Standard state

(b) Code for standard state

(c) Skipping state

(d) Code for skipping state

Fig. 7. C code emission

bound is a relevant result when evaluating the scalability of our approach.

Unfortunately, FSAs have to be emulated on real-world machines for which it is not possible to assume that all the required operations can be executed in constant time: in particular multi-way decision statements such as the switch construct are not supported natively and are transformed into multiple simpler instructions by the compiler. There are multiple alternative strategies described in the literature to implement switch operations [29] [28]: the compiler used for our tests (GCC 4.2) has been observed to use balanced trees for the very common large and sparse label sets deriving from multi-byte fields. Since the number of levels in a balanced decision tree grows logarithmically with the number of nodes, the worst-case complexity of the switch instruction is expected to be O(log N) where N is the number of switch cases. This quantity, in turn, grows roughly linearly with the number of rules when composing similar filters (recognizing e.g. TCP sessions or firewall rules that classify packets based on source/destination addresses and ports), so we expect the filter execution time to scale with the logarithm of the cardinality of the rule set.

An improvement over this O(log N) asymptotic behavior can be achieved by modifying the switch compilation strategy: as an example it is possible to use minimal perfect hashing to map case values into dense sets [30]. Perfect hashes can be expensive to compute at compile time but they execute in constant time with regard to the number of keys, lowering the asymptotic complexity of FSA-based filters to O(1). Some techniques described in the literature are explicitly aimed at supporting networking applications [31], even extending to dynamic updates.

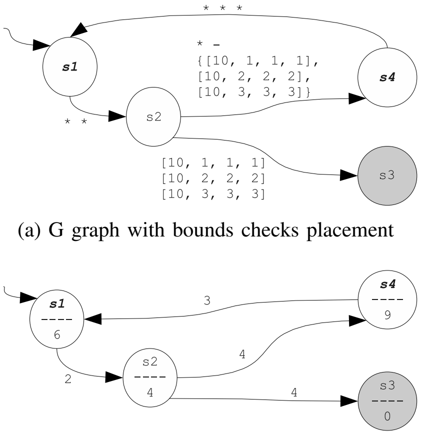
### *E. Memory access safety*

The natural solution of presenting C filter implementations with a memory buffer to hold packet data poses the problem of detecting and handling out-of-bounds accesses. A trivial and inefficient answer is to perform a comparison between the current offset and packet size on each access; better results can be obtained by reducing the number of bounds checks to be performed at run-time. In order to address this issue we have developed a *bounds checks minimisation* algorithm that places aggregate bounds checks in a small number of places in the program.

Given a compacted DFA transition graph *G*, we derive a weighted oriented graph *G'* with an edge for every DFA transition and vertex for each state; edge weights are the (positive) byte lengths of the corresponding transitions. We establish a metric on *G'* so that the distance between two states (*a*, *b*) is the shortest path from state *a* to *b*. Given this metric, the *distance from success* $d(s)$ of a vertex $s$ is the smallest distance from $s$ to a reachable, final state. If upon entering state $s$ less than $d(s)$ bytes are available in the input buffer then the filter is bound to fail as not even the nearest final state can be reached. Since each outgoing transition consumes a known amount of input data, we call $l(s, s')$ the length of the transition going from state $s$ to state $s'$: the key observation is that if $d(s) \geq d(s') + l(s, s')$ holds, then no check needs to be performed upon entering $s'$ along the transition considered.

The bounds checks minimisation algorithm works by placing a bounds check before entering the initial state (as no assumptions can be made at that point) and on all transitions that do *not* respect the aforementioned inequality, a situation arising primarily from back edges in the protocol encapsulation graph and from optional protocol parts (e.g. IPv4 options). Fig. 8a shows a DFA and Fig. 8b shows the corresponding $G'$ with distance-annotated edges and states. States marked in bold in Fig. 8a require bounds checks on at least one of their input transitions.

The bounds checks placed in the code by this algorithm not only verify that there are enough bytes left to take the next transition but that there are enough to reach the end of the computation as well. This effect is similar to bounds checks aggregation, performed (to different degrees) by general-purpose bounds check optimizers [32] and DPF [9]; whereas aggregation usually works locally, our solution considers the whole filter, achieving a very high degree of effectiveness. As an example, a TCP session filter executes a single size check when presented with a packet with no IP options and the very common Ethernet - IP - TCP structure. This check is performed at the beginning of the filter program to detect packets that are too short to contain the minimum-sized



(a) G graph with bounds checks placement

(b) G’ graph

Fig. 8. Bound checks placement

Ethernet - IP - TCP header sequence. Placing memory checks as early as possible has also the nice side effect of discarding truncated packets without having to fully decode them.

### *F. Termination in ’C’ language implementation of SPAF filters*

The FSA model clearly dictates the worst-case termination condition for any automata, which is exhaustion of the input string; filter automata respect this behavior by terminating as soon as their finite-sized input stream is completely processed, regardless of the presence of any loops in the FSA transition graph. It must be shown, however, that the C code implementation itself behaves in the same way.

The current implementation exploits bounds checks to enforce filter termination as well. A filter function returns either if a sink state[11] is reached (whether final or not) or if a memory check fails (in this case the result is always a mismatch)[12]. Backward jumps in the code, a potential source of infinite loops, pose no problem: each transition of a deterministic automaton consumes (at least) one byte; in the C implementation, this translates into the memory offset pointer being incremented in a strictly monotonic fashion. This implies that any finite input sequence will be completely consumed after a finite amount of state transitions; any further read from memory will trigger a failing bound check, thus proving that any filter eventually terminates.

## V. Experimental evaluation

In order to validate the SPAF approach, it has been experimentally compared with a set of other techniques representative of the current state of the art.

The first alternative considered is BPF, still one of the most common approaches to packet filtering; in order to avoid interpretation overheads we have used the modern JIT version described in [5]. The second selected technique is BPF+[13], which is commonly regarded as one of the most modern CFG-based approaches and addresses some shortcomings associated to other state-of-the-art generators such as PathFinder. The native UltraSparc BPF+ back-end has been modified to generate C code in order to make it compatible with the test platform. NetVM-based filters have also been selected because they are based on NetPDL protocol descriptions, therefore achieving a level of expressiveness very similar to our approach, with which NetPDL descriptions are shared. At the time of the tests being run, the NetVM did not provide any safety enforcement, neither for termination nor for memory. Finally, Ruler filters have been selected because they employ a FSA-based approach similar to SPAF. Ruler can use two code emission strategies, either walking through a memory table or emitting a C code snippet for every state: the latter was chosen as it is faster and it provides a better match to our approach, even if Ruler does not implement the optimizations supported by SPAF. For the comparison filtering rules and protocol databases (where applicable) were made as similar as possible across all the generators. As an exception, loops and multiple encapsulations were included both in SPAF and NetVM filters where meaningful, as noted; while theoretically capable of the same, Ruler uses simplified protocol descriptions throughout all the tests as there are no practical ways to specify those features with its source language.

To ensure significance, the test filters were run independently in a test bench that measures run times either with the RDTSC instruction or the `gettimeofday` POSIX system call for longer periods (more than a second). The hardware platform used for all the tests is a Dell workstation with an Intel E8400 Core 2 Duo dual-core processor with 4 GiB of RAM, running a 32-bit OS based on the Linux 2.6.24 kernel. C code was compiled with GCC 4.2. All filter processes were bound to a single processor and the machine was otherwise unloaded. Data were collected with hot disk and processor caches.

### *A. Worst-case filter performance*

The first test series aims at evaluating the emitted code quality counting the clock cycles required to execute the worst-case path through simple filters, reported in Table I. Since we are interested in comparing filters recognizing identical sets of packets, in this test the NetPDL database used for both the NetVM and the FSA-based approach was reduced to contain only the complete descriptions of the protocols involved. All the aforementioned generators were tested apart from BPF+: since no JIT emitter for the x86 platform is available it would have been hard to separate optimizations introduced by the C compiler from optimizations performed by the filter generator itself. For each generator and each filter the resulting machine code was examined and an ad-hoc packet was forged to make the worst-case code path to be executed. The test packets did not contain multiple levels of encapsulation because they cannot be handled by some approaches. BPF and NetVM measurements refer only to the proper filtering code while the cost of the respective VM frameworks has been removed.

Test results are reported in Fig. 9. The two columns “SPAF, check” and “SPAF, nocheck” refer to FSA-based filters

[11] A sink state is any state with outgoing transitions to itself for any input symbol.

[12] Any path in the FSA graph always terminates with a sink state by construction: during compilation any states with no successors are automatically linked to a non-final sink.

[13] The authors would like to thank dr. Begel who kindly provided the BPF+ source code.

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 udp.sport == 20 and udp.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` |

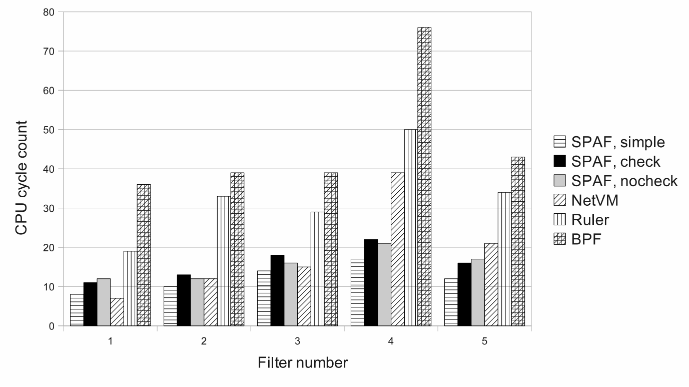


Fig. 9. Worst case filter evaluation

compiled with the aforementioned NetPDL descriptions and bound checks enabled or disabled, respectively. The "SPAF, simple" series was generated with a minimal NetPDL database designed to match as close as possible the capabilities of BPF. Fig. 9 shows that the worst-case behavior and the code quality of FSA-based programs is similar or better than with other approaches for filters of varying complexity. The best results are achieved in tests 4 and 5 where the statement is more complex because the FSA model is able to organize user-specified predicates to avoid performing redundant checks; the trend is similar for Ruler filters that nevertheless are slower, due to a less optimized emission technique that leads to a higher amount of comparisons and memory accesses to be performed. Test cases 1, 2 and 3 show that SPAF generates fast filters for low-complexity rules as well, even if the used protocol database contains conditions that the other approaches do not handle; results are further improved when considering the "simple" database.

This benchmark also shows that safety checks cause very low run-time overhead; this is fully justified by their reduction to a single comparison at the beginning of the filter; the resulting branch is made even less expensive by the branch predictor of the host CPU. It can be noted that in test cases 1 and 5 enabling safety checks actually lowers filter execution time: this apparently strange behavior is retained even when tests are repeated for a large number of times and might be caused by instruction reordering or other pipeline issues in the processor, or, given the very small measured difference (around 1 clock cycle) to unforeseen sources of error in the measurement procedure.

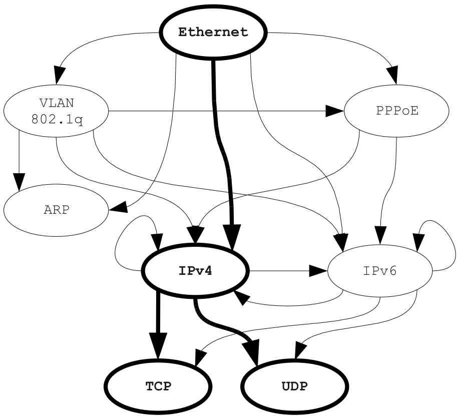


Fig. 10. Protocol encapsulation graph

### B. Filter scalability

The second test series is designed to evaluate the filtering techniques in a realistic scenario that highlights filter scalability. The rule set was created by extracting the N most active (in terms of packet count) TCP sessions from a 1 GiB real-world packet trace; packets were filtered by increasing the number of sessions to be recognized from 1 to 128. Given its size, the trace fits into the disk cache provided by the Linux operating system.

The results are presented in Fig. 11, which reports the average frame rates measured. While one-off compilation times are not considered, in order to meet the goal of providing a realistic comparison among multiple packet filtering techniques we decided not to remove the time spent in frameworks or other ancillary but essential tasks. Under this testing method NetVM filters become the slowest of the group because of the overhead introduced by the virtual machine. Their measured results are not reported in order to preserve figure readability.

The NetPDL database used for this test is represented by the encapsulation graph in Fig. 10, which includes recursive encapsulations. The graph is larger than what can be examined using BPF-derived techniques and allows SPAF to recognize TCP sessions even if the IP header is encapsulated in other protocols. This causes some protocols that do not appear in filter statements (e.g. VLAN and IPv6) to be present in the executable code because their traversal could be required to reach IP and TCP headers; in turn, this causes more operations to be performed at run-time than required by the other approaches. This difference is small but justifies some of the overhead encountered, especially with low session counts.

Fig. 11 shows that FSA-based filters scale significantly better with increasing session counts than the other techniques considered, following a logarithmic curve instead of the linear one of traditional approaches. This is expected from the theoretical considerations reported in Section IV-D. Other approaches, such as BPF+, provide lower overheads when filtering few sessions because of the smaller set of protocols and protocol features analyzed; nevertheless they scale worse than SPAF, with the crossover point being at around 10-16 sessions. In spite of the simpler protocol database, Ruler filters are slower than their SPAF counterparts, once again because

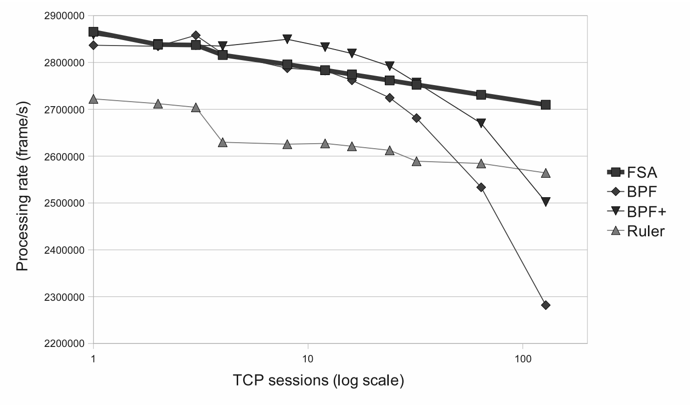


Fig. 11. TCP session filtering performance

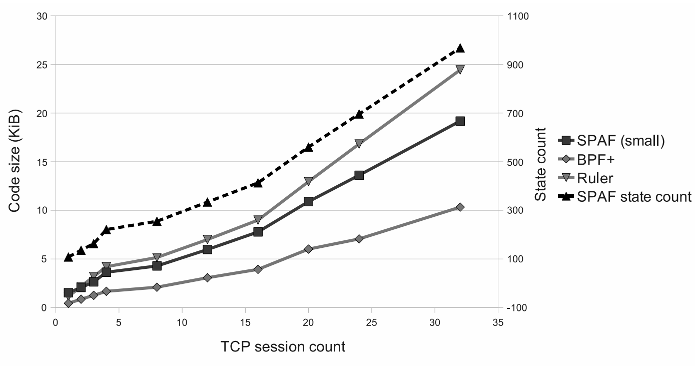


Fig. 12. Memory occupation of TCP session filters

the absence of emission-time optimizations leads to a higher number of memory accesses and comparisons to be performed. It should also be noted that SPAF absolute frame rates are good, since even in complex cases it is possible, on the test machine, to filter packets at roughly 150% the rate required for gigabit Ethernet, even if the time spent in fetching packets from memory is included in the count.

### *C. Memory consumption and potential state-space explosion*

A very felt problem with finite automata is the exponential explosion in the number of states experienced when transforming a NFA into a DFA. This issue comes from the intrinsic inability of DFAs to cope with certain pattern sets that are frequently encountered in real world applications such as intrusion detection systems [33] [17] [16]. In the general case, a NFA of $n$ states can lead to a DFA of O($2^n$) states upon determinization; the additional states introduced are not redundant and cannot be removed upon minimisation.

In order to investigate the occurrence of this issue in our context we have taken memory occupation measurements for filter statements and protocol databases of increasing complexity. Since no memory is allocated at run-time, except for a very small and fixed stack space, measurements were performed directly on the executable code with the POSIX `nm` command; Ruler and BPF+ filters are included as they are compiled to object code in a similar fashion.

Scalability w.r.t. filter rule complexity has been evaluated with TCP session filters compiled with a reduced protocol database that mimics the one supported by BPF+; more specifically, we have included all the protocols and all the encapsulation relationships shown in bold in Fig. 10. Results presented in Fig. 12 show that memory consumption is broadly linear in the number of filtered TCP sessions with no state explosion occurring; moreover, the trend of both Ruler and SPAF object files is similar, as expected. Finally, the absolute code size is reasonably small, so even big filters can fit into modern processor caches. The comparison with BPF+ shows that SPAF needs roughly twice the space; this is a good result especially when considering that, as explained in previous sections, the plain FSAs we use contain repeated portions that cause additional memory consumption. When compared with Ruler, SPAF filters use less space thanks to the automata compression technique employed.

TABLE II
MEMORY OCCUPATION OF PROTOCOL DATABASES

| Database | Protocols | Raw state count | Compressed state count |
|---|---|---|---|
| simple | `Ethernet`, `TCP`, `IPv4` (no options, no recursive encap.) | 40 | 11 |
| medium | `Ethernet`, `VLAN`, `PPPoE`, `IPv4` (no recursive encap.), `TCP` | 183 | 55 |
| complex | `Ethernet`, `VLAN`, `PPPoE`, `MPLS` (no recursive encap.), `IPv4`, `TCP`, `UDP` | 387 | 113 |
| full | `Ethernet`, `VLAN`, `PPPoE`, `MPLS`, `IPv4`, `IPv6`, `TCP`, `UDP` | 5004 | 705 |

A second test involved compiling the protocol databases reported in the second column of Table II with the filter `ethernet.src == 00:11:22:33:44:55 and ip.src == 11.22.33.44 and tcp.sport == 80`. If not differently specified, full encapsulation relationships were used: this means that e.g. TCP can follow either IPv4 or IPv6, and the whole set of protocols is actually present in the resulting automaton. The results of the second test are reported in the third and fourth column of the same table and show the state count before and after the succeed-early and the automata compression algorithms. While some protocols (such as IPv6) clearly require a larger amount of states to be represented, the final result is still manageable, especially after compression. It must also be kept in mind that the cost of the protocol database is basically fixed and largely independent from the size of the filter rule set: in the case of IPv6 the bulk of the added states is used to parse the extension headers and is not further replicated in the final automaton.

While experimental tests can provide only empirical evidence and it is always possible to design protocols and filter rules to cause an explosion in state space, we do not expect any from real-world protocol sets and filter rules. This is because filters are not as free-form as regular expressions matching arbitrary text fragments can be. As an example, finding a $.*$ pattern (widely known to be as a source of problems [17]) in packet filters is exceedingly rare as it would correspond to a field of unlimited size, while network protocols are mainly described in terms of fixed sequences of fixed-length fields; even when repetitions or variable-sized fields are encountered their maximum size is constrained by previously-read data or by maximum packet size; moreover, in general each subpattern can start in many positions in the input string, leading to a large amount of replication required to keep track of all the possible advancements in each rule. On the contrary, packet filters subpatterns (i.e. specific protocols, generic or specialized) invariably begin in a set of well-defined positions (i.e. after the previous protocol is fully parsed), thus negating another common source of space state explosion.

### D. Filter compilation time

Optimizing the generator (versus optimizing the resulting filters) was not a main objective for this work; however for completeness we report some data on filter compilation times.

Under our current implementation generation times can vary from seconds (for small protocol databases and simple filter statements) to several hours (e.g. for the full protocol database and a 128 TCP session filter). We speculate that these figures can be improved by orders of magnitude as the generator was designed to trade off performance for memory savings; moreover profiling reports of the current Java implementation show that a relevant amount of time is spent in performing tasks such as garbage collection: a different language (such as C++) is likely to reduce the associated overhead. It is also possible to employ more optimized algorithms (e.g. by performing determinization and minimisation in a single pass) that were left out due to the aforementioned choice in trade-offs.

## VI. Conclusions and future works

We have designed, prototyped and evaluated SPAF, a packet filter generator based on the creation of Finite-State Automata from a high-level protocol format database and filter predicates. SPAF aims at emitting fast and efficient filters while preserving all the relevant safety properties, both in terms of memory access correctness and termination.

The FSA model is particularly valuable for this purpose because it is powerful enough to express any possible stateless packet filter, even if containing advanced constructs not supported by most other approaches such as tunneling, multiple encapsulations and repeated header portions. In FSA filters each header field is examined at most once by construction; this property alone greatly limits the amount of redundancy in predicate evaluation, a major source of inefficiency in packet filters. On the contrary, traditional, optimization-based approaches cannot, by their own nature, provide any hard guarantee about the resulting code quality. SPAF is also able to handle filter composition trivially, using well-defined automata boolean operations; no sequentialization of multiple filters or resorting to heuristics is required. Besides being a non-ambiguous semantic model for packet filters, FSAs also have straightforward and fast software and hardware implementations, thus doubling as a guideline for efficient code emission.

In order to prove this technique on the field we have developed a filter generator that creates filters from an external protocol database and user-specified rules. Filter DFAs can be used as they are by existing hardware or software engines, or translated into C code by the back-end. We also developed an ad-hoc DFA execution engine that adapts its operations to the word size of the underlying machine instead of processing a byte at a time and enforces memory safety and termination through run-time fully aggregated bound checks.

The run-time performance and memory occupation of SPAF filters have been evaluated both in synthetic and real-world benchmarks. Test results show that FSA-based filters perform on a similar or improved level as other modern approaches such as BPF+, both on simple and complex filters; SPAF filters are also shown to scale better with increasing numbers of filtering rules. The measured overhead of run-time safety checks is small and does not to cause any significant penalties both in times of run-times (few checks are executed per packet) and memory occupation (few checks are inserted per filter). Overall, the SPAF approach is an effective and simple way to generate packet filters that are easy to compose and efficient to run, even with increasing complexity.

Among the potential problems, a widely-known issue affecting specifically DFAs is an explosion occurring in the state space when treating certain critical patterns; this problem is the limiting factor for DFA adoption in other pattern-based detectors such as intrusion detection systems. Even if in line of principle the same could be artificially triggered in SPAF filters, we believe it is unlikely to happen in practice because of the structure of real-world protocol headers. Experimental results show that under realistic conditions memory occupation grows regularly, with large filters and full protocol databases remaining tractable.

The SPAF approach can be easily extended to perform packet demultiplexing in addition to packet filtering. This is partially supported by our current generator by labeling final states with identifiers of the matching filtering rules; full support would require dynamic automata creation and code generation, tasks that will be the object of future studies. Another future extension to SPAF could be enabling interactions (e.g. look-ups and updates) with stateful constructs such as session tables, useful for higher-layer filtering and traffic classification.

In conclusion, SPAF has been shown as an approach that improves the state of the art by generating packet filters that combine most of the desired properties in terms of processing speed, memory consumption, flexibility and simplicity in specifying protocol formats and filtering rules, effective filter composition and low run-time overhead for safety enforcement. The development of the filter generator and the test results support the viability of our claims.

## References

[1] S. McCanne and V. Jacobson, "The BSD packet filter: a new architecture for user-level packet capture," in *proceedings of USENIX '93*, 1993.

[2] A. Begel, S. McCanne, and S. L. Graham, "BPF+: exploiting global data-flow optimization in a generalized packet filter architecture," *SIGCOMM Comput. Commun. Rev.*, vol. 29, no. 4, pp. 123–134, 1999.

[3] M. L. Bailey, B. Gopal, M. A. Pagels, L. L. Peterson, and P. Sarkar, "PathFinder: A pattern-based packet classifier," in *Operating Systems Design and Implementation*, 1994, pp. 115–123.

[4] O. Morandi, F. Risso, M. Baldi, and A. Baldini, “Enabling flexible packet filtering through dynamic code generation,” *proceedings of ICC ’08*, May 2008.

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

[6] S. Ioannidis and K. G. Anagnostakis, “xPF: Packet filtering for low-cost network monitoring,” in *proceedings of HPSR ’02*, 2002, pp. 121–126.

[7] F. Risso and M. Baldi, “NetPDL: an extensible XML-based language for packet header description,” *Comput. Netw.*, vol. 50, no. 5, pp. 688–706, 2006.

[8] L. Degioanni, M. Baldi, D. Buffa, F. Risso, F. Stirano, and G. Varenni, “Network virtual machine (NetVM): a new architecture for efficient and portable packet processing applications,” *proceedings of the 8th International Conference on Telecommunications*, vol. 1, pp. 163–168, June 15-17, 2005.

[9] D. R. Engler and M. F. Kaashoek, “DPF: fast, flexible message demultiplexing using dynamic code generation,” in *proceedings of SIGCOMM ’96*. New York, USA: ACM, 1996, pp. 53–59.

[10] M. Jayaram, R. Cytron, D. Schmidt, and G. Varghese, “Efficient demultiplexing of network packets by automatic parsing,” in *proceedings of the Workshop on Compiler Support for System Software*, 1996.

[11] S. Kumar, J. Turner, and J. Williams, “Advanced algorithms for fast and scalable deep packet inspection,” in *proceedings of ANCS ’06*. New York, USA: ACM, 2006, pp. 81–92.

[12] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner, “Algorithms to accelerate multiple regular expressions matching for deep packet inspection,” in *proceedings of SIGCOMM ’06*. New York, USA: ACM, 2006, pp. 339–350.

[13] M. Becchi and P. Crowley, “An improved algorithm to accelerate regular expression evaluation,” in *proceedings of ANCS ’07*. New York, USA: ACM, 2007, pp. 145–154.

[14] F. Yu, Z. Chen, Y. Diao, T. V. Lakshman, and R. H. Katz, “Fast and memory-efficient regular expression matching for deep packet inspection,” in *proceedings of ANCS ’06*. New York, USA: ACM, 2006, pp. 93–102.

[15] M. Becchi and P. Crowley, “A hybrid finite automaton for practical deep packet inspection,” in *proceedings of CoNEXT ’07*. New York, USA: ACM, 2007, pp. 1–12.

[16] R. Smith, C. Estan, and S. Jha, “XFA: Faster signature matching with extended automata,” *Security and Privacy, IEEE Symposium on*, pp. 187–201, 2008.

[17] M. Becchi, M. Franklin, and P. Crowley, “A workload for evaluating deep packet inspection architectures,” in *proceedings of the IEEE International Symposium on Workload Characterization ’08.*, Sept. 2008, pp. 79–89.

[18] T. Hruby, K. van Reeuwijk, and H. Bos, “Ruler: high-speed packet matching and rewriting on npus,” in *proceedings of ANCS ’07*. New York, USA: ACM, 2007, pp. 1–10.

[19] H. Bos, W. D. Bruijn, M. Cristea, T. Nguyen, and G. Portokalidis, “FFPF: Fairly fast packet filters,” in *Proceedings of OSDI ’04*, 2004, pp. 347–363.

[20] L. Deri, “nCap: wire-speed packet capture and transmission,” in *E2EMON ’05: Proceedings of the End-to-End Monitoring Techniques and Services on 2005. Workshop*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 47–55.

[21] Z. Wu, M. Xie, and H. Wang, “Swift: a fast dynamic packet filter,” in *proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*. Berkeley, CA, USA: USENIX Association, 2008, pp. 279–292.

[22] J. E. Hopcroft, R. Motwani, Rotwani, and J. D. Ullman, *Introduction to Automata Theory, Languages and Computability*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2000.

[23] P. Briggs and K. D. Cooper, “Effective partial redundancy elimination,” in *PLDI ’94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*. New York, USA: ACM, 1994, pp. 159–170.

[24] R. Gupta, D. A. Berson, and J. Z. Fang, “Path profile guided partial redundancy elimination using speculation,” in *proceedings of the 1998 International Conference on Computer Languages*. Washington, DC, USA: IEEE Computer Society, 1998, p. 230.

[25] V. Laurikari, “Efficient submatch addressing for regular expressions,” Master’s thesis, Helsinki University of Technology, 2001.

[26] M. Becchi and P. Crowley, “Efficient regular expression evaluation: theory to practice,” in *proceedings of ANCS ’08*. New York, USA: ACM, 2008, pp. 50–59.

[27] A. W. Appel and J. Palsberg, *Modern Compiler Implementation in Java*. New York, USA: Cambridge University Press, 2003.

[28] A. Korobeynikov, “Improving switch lowering for the LLVM compiler system,” in *Proceedings of the 2007 Spring Young Researchers Colloquium on Software Engineering*, May 2007.

[29] Úlfat Erlingsson, M. Krishnamoorthy, and T. V. Raman, “Efficient multiway radix search trees,” *Inf. Process. Lett.*, vol. 60, no. 3, pp. 115–120, 1996.

[30] D. E. Knuth, *The art of computer programming, volume 3: (2nd ed.) sorting and searching*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1998.

[31] Y. Lu and B. Prabhakar, “Perfect hashing for network applications,” in *in IEEE Symposium on Information Theory*. IEEE Press, 2006, pp. 2774–2778.

[32] T. Würthinger, C. Wimmer, and H. Mössenböck, “Array bounds check elimination for the java hotspot$^{TM}$client compiler,” in *proceedings of the 5th international symposium on Principles and practice of programming in Java*. New York, USA: ACM, 2007, pp. 125–133.

[33] S. Kumar, B. Chandrasekaran, J. Turner, and G. Varghese, “Curing regular expressions matching algorithms from insomnia, amnesia, and acalculia,” in *proceedings of ANCS ’07*. New York, USA: ACM, 2007, pp. 155–164.

**Pierluigi Rolando** (pierluigi.rolando@polito.it) received his M.Sc. in computer and system engineering from Politecnico di Torino in 2007 with a thesis on developing a compiler for a systolic network processor. He is currently a Ph.D. candidate in computer and system engineering at Politecnico di Torino. His main research area focuses on the application of formal methods to packet processing, finite state automata and special-purpose virtual machines.

**Fulvio Risso** (fulvio.risso@polito.it) received his Ph.D. in computer and system engineering from Politecnico di Torino in 2000 with a dissertation on quality of service in packet-switched networks. He is an assistant professor in the Department of Control and Computer Engineering of Politecnico di Torino. His current research activity focuses on efficient packet processing, network analysis, network monitoring, and peer-to-peer overlays. He is the author of several papers on quality of service, packet processing, network monitoring, and IPv6.

**Riccardo Sisto** (riccardo.sisto@polito.it) received the M.Sc. degree in electronic engineering in 1987, and the Ph.D degree in computer engineering in 1992, both from Politecnico di Torino, Torino, Italy. Since 1991 he has been working at Politecnico di Torino, in the Computer Engineering Department, first as a researcher, then as an associate professor and, since 2004, as a full professor of computer engineering. Since the beginning of his scientific activity, his main research interests have been in the area of formal methods, applied to software engineering, communication protocol engineering, distributed systems, and computer security. On this and related topics he has authored and co-authored more than 70 scientific papers. Dr. Sisto has been a member of the Association for Computing Machinery (ACM) since 1999.