FSA-based Packet Filters

Pierluigi Rolando, Riccardo Sisto, Fulvio Risso
Dipartimento di Automatica ed Informatica
Politecnico di Torino
Email: {pierluigi.rolando, riccardo.sisto, fulvio.risso}@polito.it

***Abstract*—We propose a packet filtering technique based on the generation and composition of Finite-State Automata (FSA), in contrast to most traditional imperative approaches. FSAs provide a formal framework with well-defined composition operations and enable the generation of optimized executable code without resorting to multiple and opportunistic optimization algorithms. Memory safety in filters is enforced with minimal run-time overhead and termination can be proved without restricting filter control flow graphs to be acyclic, thus enabling full parsing of complex protocol formats and multiple encapsulation layers.**

Experimental evidence shows that this approach is viable and improves the state of the art in terms of filter 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 [1] that can be used to classify network traffic in accordance to a set of user-provided rules; they are a basic component of many networking applications, ranging from traffic shapers and analyzers to demultiplexers, firewalls, intrusion detection systems and more.

The modern networking scenario poses multiple requirements for packet filters, mainly in terms of processing speed (to keep up with network line rates) and resource consumption (to run in constrained environments). Preserving the integrity of the execution environment is crucial, both in terms of memory access safety and termination enforcement, even more so when running in kernel space or on the bare hardware. Since proving that a generic program written for a Turing-equivalent machine stops is a notorious undecidable problem [2], many existing generators restrict filters to acyclic structure, thus making it difficult to parse packets with multiple levels of encapsulation or repeated header fields sequences. Moreover, on most platforms packets and all the related data structures are stored in random-access memory, so it is necessary to ensure that no access is ever performed beyond the established bounds. Finally, filtering techinques should also be flexible and effective in specifying predicates and protocol formats.

Existing approaches to packet filtering focus on subsets of these issues, each one providing improvement over the state of the art in terms of performance, flexibility or safety, but, to the best of our knowledge, not all at the same time. As an example, both BPF+ [3] and PathFinder [4] generate optimized and safe filters that are unable to support recursive encapsulation relationships; NetVM-based filters [5], on the contrary, can parse complex packets but no provision for enforcing termination is currently present, either in filter programs or in the underlying virtual machine.

Most of the generation techniques proposed so far address the performance issues by applying opportunistic optimizations on the generated code. This paper investigates a new technique for packet filter generation that takes a different approach and exploits the Finite State Automata (FSA) model. The technique is aimed at stateless filtering (layer 2 to layer 4) and at achieving high performance and expanded flexibility, while maintaining all the desired safety properties. Filter generation times and dynamic rule set updates are not taken into consideration. This restriction in scope excludes some potential use cases, such as stateful TCP session filtering, where the rule set should be modified on-the-fly. Nevertheless, FSA-based filters are still useful for many classic packet filter applications, such as monitoring and traffic trace filtering, where changes in the rule set occur rarely and not at run-time. We also explicitely forgo the architectural aspects of packet filtering, such as their placement in the operating system networking stack: architectural issues can be considered orthogonal to our compilation technique and it will be shown that FSA-based filters have all the safety properties required to run in kernel space, if needed.

A packet filter can be expressed as a set of predicates on packet fields, joined by boolean operators: maybe the most important question in designing generation techniques for high-performance filters is how to efficiently organize this set of predicates. Many existing filter generators [3] [5] [6] [7] [8] are based on the synthesis of imperative programs where predicates form a Control Flow Graph (CFG). This representation is quite flexible but, for the same reason, prone to redundancies that are hard to spot and optimize away. Other techniques organize predicates into regular structures such as tries or trees [9] [4]. This makes it easier to spot redundancies and other sources of overhead but might also carry limitations: it is e.g. intrinsically hard to express the cyclic relationships required to support tunneling or repeated protocol portions with acyclic structures such as trees.

The FSA approach considers packet filtering as a regular language recognition problem and it provides a simple mathematical framework to express and organize predicates. As shown in this paper, the FSA model enjoys several interesting properties, such as reducing by construction the amount of redundancy along any execution path in the program.

Automata are straightforward to translate into efficient executable forms without requiring a full-blown optimizing

compiler. Safety, both in terms of termination and memory access integrity, can be enforced with very low run-time overhead, as shown in this paper. Finally, FSAs provide a natural way to express tunneling, encapsulation relationships and repeated header portions.

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 our conclusions and also highlights possible future developments.

II. Related works

There is a large corpus of literature on packet filters. The best-known and most widely employed technique is probably BPF [7], 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. Filter predicates are structured into a control flow graph (CFG) where backward jumps are disallowed (thus forgoing support for e.g. IPv6 extension headers); memory protection is enforced by checking each access at run-time. Multiple rules can be composed together by boolean operators. In its original implementation, only a small set of optimizations were perfomed over predicates, either within a single filter or across many. The bytecode was interpreted, leading to a considerable run-time overhead impact which can be reduced by employing JIT techniques [10].

A lot of work builds on BPF and, more generally, CFG-based models, either by bringing architectural modifications or by improving the filter generator process. BPF+ [3] improves filter compilation by adding a number of local and global data-flow optimization algorithms that remove redundant operations by altering the CFG structure. The resulting program is translated into BPF bytecode, then emitted to native instructions just-in-time. While these improvements speed up filter execution w.r.t. BPF, protocol descriptions are still hardwired in the filter generator and no precautions are taken to lessen the impact of checking memory accesses at run-time. Moreover, a potential BPF+ issue is that, like in every CFG-based technique, the set of optimizations it performs are designed to handle a given set of improvement opportunities that are considered to be common. While some code transformations are clearly useful both in general terms and for the specific problem at hands, it is possible that unforseen applications or future developments introduce sources of redundancies that need additional care: no conclusions can be taken a-priori about the quality or structure of filtering code.

A more recent BPF derivative is xPF [6], whose main contributions w.r.t. filter generation are the addition of persistent storage to preserve data across multiple filter executions and allowing backward jumps in the control flow graph of the filter in order to provide relief for what is reported to be an unacceptable restriction. Termination is enforced by an execution monitor that limits the maximum number of instructions executed, an approach that is viable as long as the filter instructions are interpreted, but that might prove difficult to extend to native code emission, and the additional overhead imposed has not been measured.

A CFG-based filter generation technique unrelated to BPF is described in [5]. Its main contribution consists in decoupling the protocol database from the filter generator by employing an XML-based protocol description language, NetPDL [11]. The filter code is executed on the NetVM [12], a special-purpose virtual machine targeting network applications. The NetVM provides a JIT compiler which takes care of performing common optimizations, both on filter structure and on low-level code. A relevant result is that 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.

A different approach has been taken by PathFinder [4], which departs from the CFG model to render predicates as template masks (atoms). Each packet is scanned linearly from the beginning to the end while atoms are applied until a result is reached. Atoms are ordered in a decision tree so that shared prefixes can be coalesced and evaluated only once. It has been noted that merging only common prefixes is not sufficient to catch some common optimizations opportunities, where there are no shared prefixes but predicate evaluation is nevertheless redundant [3]. The PathFinder approach has been shown to work well both in software and in hardware but does not solve any issues related to the protocol database and memory accesses. FSA-based filters are similar to PathFinder because packet contents are processed in a similar order but without its limitations and filter composition is handled to a wider degree.

DPF [9] improves on PathFinder by generating machine code just-in-time and avoiding whenever possible the repetition of multiple memory offset checks within the same atom. Further optimizations are also added, such as using a flexible strategy to implement multi-way conditional choices; nevertheless, the redundancy elimination strategy is still based on prefix merging. DPF implements a bounds checks reduction strategy that aggregates checks by scanning each atom and checking only the availability highest memory offset required to succesfully complete the filter. The bounds checking reduction technique described in section IV-E aggregates multiple checks in a similar fashion but considers the whole filter at the same time, thus reducing run-time overhead to a single check in most cases.

A further approach has been presented by Jayaram et al. [13] that uses a pushdown automaton to perform packet demultiplexing; filter specifications are expressed as LALR(1) grammars and can be therefore effectively composed. While this is shown to greatly improve filter scalability, there are downsides related to the push-down automaton: while a naïve implementation is trivial, specific optimizations are required to

achieve good performance. The format chosen for expressing filter predicates and protocol structure, while certainly flexbile, is also quite unusual and the grammar must be kept unambiguous, a task which might prove difficult in complex situations. The authors marginally note that the simpler FSA model would be sufficient for the same task.

A differentiating point between different packet filtering approaches is how redundancies and dependencies in the filter predicate set are handled. CFG-based techniques such as BPF, BPF+ and the NetVM use optimization algorithms derived from general-purpose compilers to reorganize code to different degrees of effectivness, while PathFinder and DPF mainly employ prefix coalescing, made easier by their internal organization of filter predicate sets. Whereas BPF optimizes only protocol parsing, by avoiding to reexamine lower layer headers, PathFinder and DPF use a trie-like structure that supports prefix merging: each time a new filter is added to the active set the generator descends into the data structure until a new leaf can be appended. If no common prefix is found, a new trie must be created and the whole trie set is then examined at run-time, until a match is found. This approach works well if all filters are expressed with the same predicate order (a condition enforced by construction by DPF) and if a common prefix can be recognized. However, this is not always the case: as an example the following simple filter causes a redundant comparison to be performed:

```
  (ip source = X and tcp dport = Y) or
(tcp sport = Z and tcp dport = Y)
```

In this case a common prefix is shared between the two subexpressions up to the IP header only: if the first part fails because `tcp dport` is not `Y`, then the second portion of the filter is needlessly executed. This specific optimization opportunity is caught by the data-flow optimizations performed by BPF+, that are able to detect a potentially duplicate evaluation of the same predicate and optimize it away by transforming the filter CFG appropriately.

As it will be seen, FSA filters examine each header filter at most once by construction, because automata scan their input sequentially and the resulting executable code mimic rather closely this behaviour, and this property is preserved across filter composition operations. The technique described in this paper is therefore at least as capable at redundancy removal as the best alternatives described in literature.

Other improvements come from architectural considerations, as demonstrated for instance by xPF, FFPF [14] and nCap [15], or from supporting dynamic rule sets as with the SWIFT tool [8]. These techniques are out of scope for the purpose of this paper.

III. Filter generation technique

The filter generation technique we have developed consists in creating and implementing a Finite-State Automaton (FSA) that recognizes a user-defined set of packets.

While FSAs are normally expressed as regular expressions, the notation is unusual in the field of packet processing where filters are usually specified as predicates over header fields; for this reason we have developed a simple filter specification language, rather similar to the one used by BPF, that additionally contains some provisions to support encapsulation relationships. The input of the generator consists in a set of filter rules and a protocol database providing on-the-wire formats.

The task the filter generator must perform can then be logically divided into 3 independent steps:

1) each protocol in the externally-provided database is translated into an augmented FSA representation which will act as a template for further processing;
2) filter rules are compiled, one predicate at a time, to get specialized FSAs, each recognizing a subportion of the filter;
3) the FSAs obtained from step 2 are merged together using boolean operators.

The end result of the generation process is a Deterministic Finite-state Automaton (DFA) which is a well-defined formal description of filter semantics and, upon minimisation, is its canonical form.

The rest of this section provides a formal definition for FSAs and details over the 3 generation steps outlined above, while section IV describes the second part of the generation process, where the filter DFA is translated into executable form.

A. Finite-State Automata

A Finite-State Automaton (FSA) is a quintuple $(\Sigma, S, s_0, \theta, F)$, where Σ is an alphabet of input symbols, S is the set of states, $s_0 \in \Sigma$ an initial state, $\theta \subseteq S \times \Sigma \times S$ the transition relation and $F \subseteq S$ the set of accepting states. For our purposes, Σ is the set of all the possible 8-bit strings plus the empty symbol ϵ, used to label transitions that can be taken without consuming any input.

Finite-state automata can be used to represent regular sets of symbol strings; it is easy to check whether any given string belongs to a set represented by a FSA and regular sets are closed under all of the classical set operations (union, intersection, complementation, etc).

In order for FSAs to be applicable to the problem of packet filtering, it must be proven that any possible set of packets can be represented by an FSA, which is true iff all the sets of packets are regular. Regularity is trivially verified by noting that the set of possible packets has a finite cardinality, as for any given packet-switched network there is a maximum frame size allowed by the data-link technology. Since any finite set is regular, and because any filter recognizes a subset of all the possible packets, FSAs are a suitable analytical way to model stateless packet filters, and this provides a sound theoretical basis for our efforts.

B. Protocol database compilation

The first phase of the filter generation process consists in parsing the protocol database and generating template automata. For each protocol, the generator creates an automaton that accepts all and only the packets that respect the protocol format. These automata can be non-deterministic. During this

```
<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

phase, some automata transitions are annotated with labels that act as anchors for determining which packet field is being parsed upon reading a certain input byte. These labels are used in the next phase of the compilation process to apply filtering conditions over the automata created at this step and are discarded soon afterwards, so they do not influence in any way the FSA-handling algorithm or the properties of the model.

The protocol database is kept separated from the generator itself so that it can be freely modified. We have employed the NetPDL protocol description language [11]. NetPDL is an XML-based language to describe the on-the-wire structure of a network protocol and to specify encapsulation relationships. This allows to parse input packets up to layer 7. NetPDL provides primitives to describe protocol fields of both fixed and varying size either expressed in bits or bytes. More complex structures such as optional or repeated sections can be expressed by a number of control-flow-like primitives that include conditional choices and loops. Encapsulations are described by predicating conditions over one or multiple protocol fields. A simplified NetPDL description of the IPv6 header format is presented in fig. 1.

Our filter generator reads its protocol database from an external NetPDL source, so it is effectively protocol-agnostic: a new protocol requires only its NetPDL description to be supported in the generator and its fields are automatically made available in the filtering language as well.

In its current form, the FSA generator supports a proper subset of the NetPDL primitives, which can be used to describe most existing layer 2 to layer 4 protocols. The excluded constructs, such as stateful session tables, target higher layers or do not match well the capabilities of the FSA model. FSA generation has been succesfully tested on the full versions of the most common 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.

As long as it is correctly performed, the creation of FSAs from NetPDL descriptions is not critical to filtering performance, as it will become clear from later sections; its exact inner working can be therefore regarded as a volatile implementation detail, prone to improvements in future versions of the filter generator. Nevertheless, in order to provide the reader with some examples, we report the automata generated for a simple fixed-length header field (fig. 2b), a 2-way conditional construct (fig. 2d) and a more complex situation where a loop is interlocked with a `switch` construct, closely representing the structure of the IPv6 extension headers (fig. 2f). Internal field names are reported where relevant.

As it can be seen from the figure, simple NetPDL constructs such as fixed field have a straightforward FSA translation. Other constructs, such as `switch`, need more attention: since the FSA model has no concept of discrete storage locations but it is nevertheless necessary to remember values read previously for subsequent computations, a number of parallel branches has to be spawned within the automaton. This often leads to the replication of some automaton portions.

The automaton in fig. 2b can be easily related to the NetPDL description in fig. 2a as it simply consists in a transition chain that consumes as many symbols as the length of the `type` field. Fig. 2d presents a 2-way conditional construct. The choice between blocks `B` and `C` (that abstract automaton portions) depends on the value of filed `type`; since `A` stands in the path between the point where `type` is read and the point where its value is actually needed, it needs to be replicated. In any case, at run-time only one copy of `A` will actually be executed. Finally, fig. 2f shows a more complex case where a `switch` construct similar to the one in fig. 2d is embedded in a loop. This time the field involved in the `switch` construct is read at iteration N but affects the outcome of iteration N+1 so the transition graph is more complicated. Both the block `A` and the states implementing the inner fields must be replicated in this case.

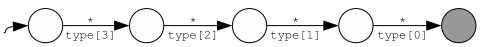
C. Filter rule imposition

In the second processing step filtering rules are parsed into blocks i.e. atomic units predicating one or multiple conditions over a single protocol header that must all be true at the same time. In a single rule there may be one or more blocks, referring to one or multiple protocols (multiple blocks referring to the same protocol are allowed), joined together with boolean operations. A simple filter statement such as `ip.src = X and tcp.sport = Y` is therefore divided into two blocks, that are both operands for the boolean `and` operator. Filtering conditions are then imposed over template FSAs one block at a time, thus creating a set of FSAs, each one recognizing all the packets with correctly formed protocol headers and matching the condition of the block involved.

In order to apply the conditions predicated by a block to a protocol, the filter generator uses the annotations on template FSAs to find the transitions consuming the desired bytes of the specified field, then replaces them with specialized

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

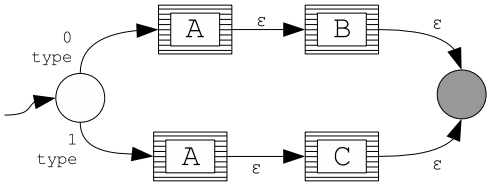
(a) Fixed field



(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>
```

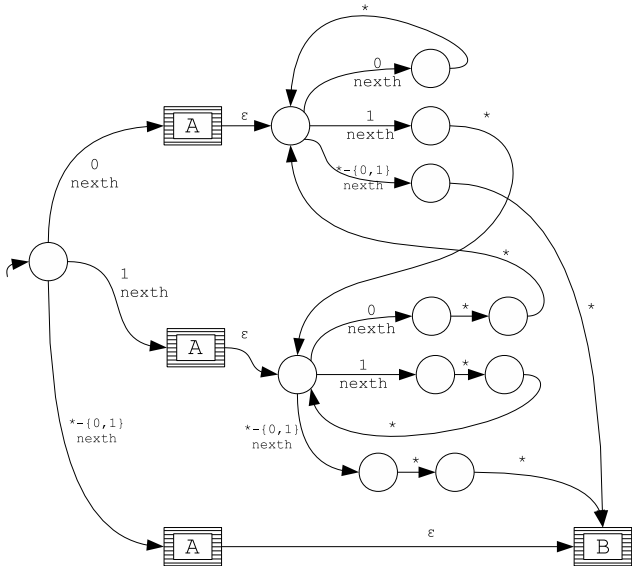
(c) Switch construct



(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

versions that trigger only upon matching the requested value. Particular care must be taken when this mechanism interacts with optional (or repeated) protocol parts and, in general, when fields used in the NetPDL database (e.g. IPv4 header length) are also subject to user-specified predicates: in these cases the automaton structure might need to be modified to accomodate the transition specialization. In the aforementioned filter, the generator would create a specialized copy of the IP automaton that only accepts packets with the correct IP address, then a copy of the TCP automaton to recognize headers with the source port equal to Y.

D. Filter composition

In this step the FSAs created previously are joined together in order to build the final filter automaton. This operation is performed in the order dictated by the filter statement and uses well-known algorithms [16] to implement boolean operations and make the resulting automaton both deterministic and minimal.

Up to this step, the FSAs used in the generator do not necessarily conform to any specific requirements; in particular they can be indifferently deterministic or non-deterministic. However, to our knowledge there are no algorithms to complement non-deterministic FSAs, so a determinisation step is needed beforehands.

Automata theory shows that a minimised DFA is unique for a given filter statement and protocol database, regardless of the compilation process: even if a totally different procedure were used, the final result would still be the same, the main difference being space and time requirements for compilation. This provides the theoretical basis for regarding the whole compilation process as described so far as an implementation detail; the current prototype shows that there is at least one practical way to perform filter generation. In a similar way, the FSA-based approach is also insensitive to any specific predicate order or other peculiarities of the filtering rules: the final result is dependent on the semantics of the filter statement only, regardless of its syntactic form.

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 filter DFA can be interpreted both as a semantic model of the filter that univocally identifies the recognized packet set and as a guideline for a software implementation that closely mimics the mathematical behaviour of the automaton. Having a machine-agnostic representation can be useful because it decouples the generation process from the code emission routines and allows porting the filter across platforms while fully preserving its semantics.

Another very useful property of FSA-based filters is that their construction process ensures that each field of the packet is examined at most once, no matter how complex the protocol database or the filtering rules. Since automata are closed under boolean operators this property is preserved also across compositions of any nature. This property is useful because it provides a trivial method for the code generator to avoid redundant comparisons and makes each execution path in the filter quite fast. There are downsides to this built-in efficiency, however. In the FSA model, filters are by design limited to examining packet fields in the same order as they appear on the wire. Since the current code emission technique closely mimics the behaviour dictated by the model, 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 constitues a form of partial redundancy that can be handled by many algorithms described in literature [17] [3] [18]. In spite of these considerations, good quality code can be generated even if this issue is ignored, as the number of partially redundant checks is quite low when considering real-world protocols.

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 architecture, 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 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[1] or the PCRE library[2]. In practice, however, our scope and specific requirements are sufficiently different from the mainstream application of regular expressions to justify the development of an ad-hoc engine which can be made simpler (as no advanced features such as backtracking are required) and faster (by performing DFA transformations that would not be useful in the general case). Generic regex engines usually implement features such as backtracking or subexpression matching that are useful in a general-purpose tool but are not needed for our goals. Supporting these features creates run-time overhead that we can easily avoid [19].

A DFA stops when the input symbol sequence is over; this behaviour should be emulated by the software implementation, that however receives a memory buffer and not a data stream as its input: the stream semantics should be mimicked using as few run-time operations as possible. A naïve implementation that performs a termination check at each computation step is likely to be too expensive. 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.

The rest of this section describes the transformations performed over the filter DFA in order to improve 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

In many practical instances, filters are not used to verify packet well-formedness but only classify network traffic according to some user-specified conditions. Under this assumption, it makes sense to forgo some protocol header format requirements imposed by the protocol database and terminate the filter as soon as the user-specified rules are matched. As an example, if the user wishes to filter packets based on their IP source address only, it might make sense to stop as soon as the IP source address field is encountered.

The filter generator can run an optional algorithm that improves peformance in these cases by removing a trailing portion of the filter, after all user-specified rules have been matched.

The effects of this *succeed-early* algorithm are shown in fig. 3, where an automaton is simplified by coalescing all the states post-dominated by the final one.

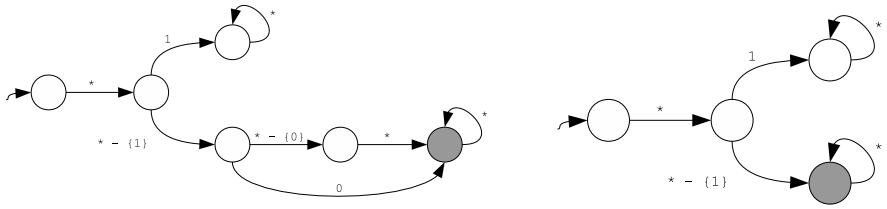
The succeed-early algorithm modifies the packet set recognized by the filter: as an example, fig. 3b no longer contains the well-formedness check dictating that a non-zero byte should be followed by a non-empty byte string. More specifically, the algorithm removes all the states along any path that would lead to success if enough input symbols were available. This makes the filter cheaper to execute, as in the example one less memory read and conditional choice are required, but also extends the filter to recognize truncated packets. 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 a waste of processor cycles and memory bandwidth that should be avoided whenever possible. In order to improve performance, the generator searches DFA transition graphs

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

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



(a) Original automaton (b) After the succeed-early algorithm

Fig. 3. Succeed-early algorithm

for sequences of byte-skipping transitions, visually marked with stars in figures; whenever possible, these are compacted into a single one, marked with the correct length, and all the intermediate states are removed. This constitues the *star compaction* algorithm.

A similar optimization is perfomed on non-star transitions: while our filter DFA works natively on 8-bit values, most CPUs are capable of processing more than one byte at a time, making reapeated single-byte operations more expensive that fewer multi-byte ones. 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. In line of principle transition compaction is performed similarly to star compaction: starting from a single state, the transition graph is explored to build long chains of transition to be replaced with a multi-byte one. In contrast to star compaction, however, the maximum number of transitions to be coalesced is limited to the machine word size; furthermore, the maximum amount of transitions created by the algorithm is limited to avoid a potentially exponential explosion.

Neither star compaction nor transition compaction remove any paths in the FSA graph, so the language (and thus the set of packets) recognized by the filter is left unmodified. As an example, we have reported a sample automaton both before (fig. 4a) and after (fig. 4b) its star and transition compaction.

The combined effect of transition merging is somewhat similar to DFA multi-striding [20]. A relevant difference is that multi-striding keeps all transitittions of the same length, so the resulting object is still an automaton with a different input alphabet; on the contrary, the transition merging pass produces transitions of different sizes[3] and the result cannot be regarded as an automaton, as there is no well-defined input alphabet any longer. This creates no issues as no further automata operations need to be performed after the compaction step in the compilation process; dropping the requirement of making every transition of the same length also provides additional flexibility because it allows to compress only local portions of the transition graph, 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 consists in 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 full-fledged JIT compiler for the direct emission of assembly code for any physical or virtual machine would not be difficult to build, if needed.

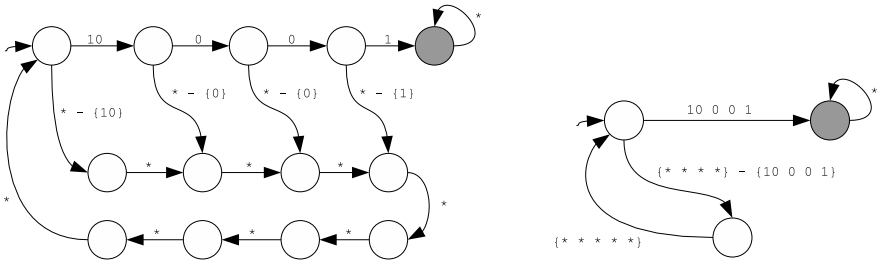
Given the relatively simple structure and behaviour of DFAs, there are multiple possible software implementations. Perhaps the most straightforward automaton implementation consists in keeping an explicit transition table in memory, but this might require many memory accesses with hard to predict patterns to walk through the automaton graph. Our generator emits a code snippet for each state and its outgoing transitions: this approach enables a better exploitation of the CPU prefetch and branch prediction units.

The required execution environment is minimal, providing as filter input a memory buffer that holds the packet and its length. No other facilities (e.g. memory protection or external libraries) are needed. The C filtering function is made up of instruction blocks, one per state, each uniquely identified by a label and containing the instructions used to compute the next state that should be reached.

Given that the input stream is replaced by the aforementioned packet buffer, it is necessary, in the general case, to execute the following steps:

- read the required amount of input bytes. This is taken care of by fetching the correct amount of bytes from memory into an appropriately-sized variable;
- increment the memory offset pointer by the amount of bytes read;
- perform a multi-way conditional comparison with the patterns derived from outgoing transition labels of the DFA model by using a `switch` statement;
- jump to the correct next state by using a `goto` statement.

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



(a) Original automaton (b) After the compaction pass

Fig. 4. Automata compaction algorithm

As an exception to this behaviour, states with only a byte-skipping outgoing transition can be implemented by simply increasing the offset pointer by a fixed amount and jumping to the following state. No packet reads are needed and no `switch` construct is required. A code emission sample for both cases is depicted in fig. 5.

There are no other essential primitives, and all byte-swapping and arithmetic operations can be performed at compile-time. A possible optimization is to employ arithmetic operations to reduce the cardinality of multi-way statements. As an example, the IP length header check is currently translated into an 11-way conditional construct with 16 labels in each case[4]; the same operation could be replaced with a more traditional masking of the lower 4 bits of the field.

Once the C function is generated, it can be fed into a C compiler to be translated into executable code and then linked to more complex applications.

Even with compiler optimizations enabled, the C code is not very prone to transformations because of the properties directly derived from the FSA model. In particular, along any possible execution path each comparison is relevant for the final result (so it cannot be optimized away) and any memory read is useful as well. Since no arithmetic operations are performed, apart from incrementing the offset pointer by a constant value, the impact of any related optimization algorithm is expected to be very small. Similarly, loops are not unrolled or modified because all exit conditions depend on packet data, which is not available 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 [21] and choosing a good switch emission strategy [22]. In particular the `switch` emission strategy is critical, because it is an operation very common in FSA-based filters, and it is not natively implemented by most processing architectures. The next section explores this problem further.

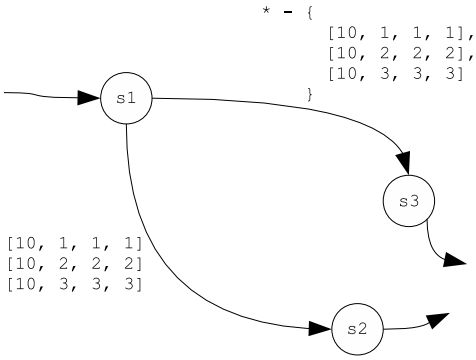
[4]These figures derive directly from the IPv4 protocol definition: header length is a count of 32-bit words, stored as the lower 4 bits of a byte. Some values are invalid, as the minimum IP header length is 20 bytes.

D. Asymptotical complexity of FSA-based filters

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, independently from the complexity of both the rule set and the protocol database. This would not provide any warranties over the actual speed of FSA-based filters, as constant and multiplicative factors might still be large; this concern is addressed in section V. Nevertheless, this asymptotical 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 natively supported and must be transformed into multiple simpler instructions by the compiler. There are multiple alternative strategies described in the literature to implement `switch` operations [23] [22]: the compiler used for our tests (GCC 4.2) has been observed to use binary decision trees for the large, non-dense case sets that are commonly encountered with 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 address and service), so, with the current compilation strategy, we expect the filter execution time to scale as O(log M) where M is the number of filter rules employed.

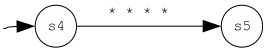
An improvement over this O(log M) asymptotical behaviour can be achieved by modifying the `switch` compilation strategy: as an example it is possible to use minimal perfect hashing, where hashing functions are used to map case values into dense sets [24] [25]. Perfect hashes can be expensive to compute at compile time but they execute in constant time



(a) Standard state

```
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;
      }
```

(b) Code for standard state



(c) Skipping state

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

(d) Code for skipping state

Fig. 5. C code emission

with regard to the number of keys, lowering the asymptotic complexity of FSA-based filters to O(1). Among the others, some techniques described in the literature are explicitly aimed at supporting real-time updates in networking applications [26].

E. Memory access safety

While the FSA model assumes that input comes as a stream of symbols, it is more natural for software implementations to present the C program with a memory buffer that holds packet data. This poses the problem of detecting and handling out-of-bounds accesses to the buffer; performing a comparison between the current offset and packet size upon each access is an effective but expensive solution to this issue which can be improved by reducing the number of bounds checks to be performed at run-time.

In order to address this issue we have developed the *bounds checking minimisation* algorithm that places aggregate bounds checks in a small number of places in the program, thus reducing run-time overhead while preserving memory safety.

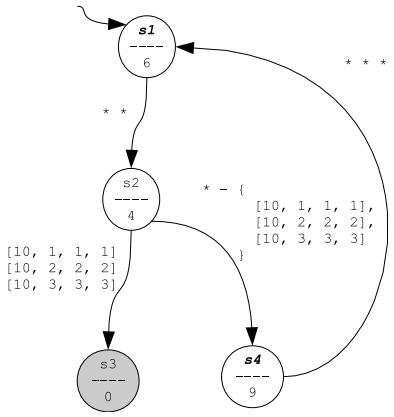
Given a compacted DFA transition graph *G*, we derive a weighted oriented graph *G'* with an edge for every DFA transition and vertex for each state; edge weights are the (positive) byte lengths of the corresponding transitions. We establish a metric on *G'* so that the distance between two states (*a*, *b*) is the shortest path from state *a* to *b*. This metric is used to compute the distance of every vertex from the nearest one that corresponds to a final state; we call this *distance from success* and denote it as $d(s)$ for any state s. If, upon entering state s, less than $d(s)$ input bytes remain, then the filter is bound to fail as not even the nearest final state can be reached. Additionally, since transitions consume a known amount of input data, we can denote as $l(a, b)$ the length of a transition going from state a to state b: if $d(a) \geq d(b)+l(a, b)$, then no check needs to be performed upon taking the transition considered.

The bounds checks minimisation algorithm works by placing a bounds check before entering the initial state (no assumptions can be made at that point), then placing bounds checks on all transitions that do not respect the aformetioned inequality. These transitions derive mostly from back edges in the protocol encapsulation graph and optional protocol parts (e.g. IPv4 options). Fig. 6a shows a DFA where each state is annotated with its distance from success, and fig. 6b shows the corresponding G' with distance-annotated edges. States marked in bold in fig. 6a require bounds checks on at least one of their input transitions.

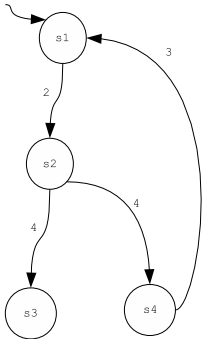
With the bounds check minimisation algorithm each check does not only verify that there are enough bytes left to take the next transition, but that there are enough to reach the end of the computation as well. This effect is similar to bounds checks aggregation (or grouping), which is performed (to different degrees) by general-purpose bounds check optimizers [27] and DPF [9]. While aggregation often operates on local opportunities, the bounds check considers the whole filter, achieving a high degree of effectiveness: as an example, a TCP session filter requires exactly one size check when parsing a plain packet with no IP options and common encapsulations. This check is performed at the beginning of the filter to detect packets that are too short to contain the minimum-sized Ethernet plus IP plus TCP header sequence. Placing memory checks as early has possible has also the nice side effect of discarding truncated packets without having to fully decode them.

F. Termination in C FSA-based filters

The FSA model clearly dictates the worst-case termination condition for any automata, which is exhaustion of the input string. This provides the theoretical ground for proving termination for any filter automaton: it terminates as soon as its finite-sized input stream is completely processed, regardless of the presence of any loops in the FSA transition graph. The



(a) G graph with bounds checks placement



(b) G' graph

same property, however, must be shared by the C code implementation, a requirement not trivially fulfilled: the currently implemented back-end treats termination and bound checking as intertwined problems, by carefully placing bounds checks and exploiting those to enforce filter termination as well.

Given the current back-end, the filter function returns in the following cases:

- a sink state is reached, whether final or non-final (sinks have only one outgoing transition, leading to themselves);
- a memory check fails (in this case the filter returns with a mismatch).

It must be noted that 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.

Backward jumps in the code derive from loops in the FSA transition graph, but it must be taken into account that the generator uses detereministic automata for code emission. In (compacted) DFAs each transition consumes (at least) one byte and there can be no ϵ-transitions: translated to C code, this means that the offset pointer is strictly monotonically incremented. In turn, this implies that any finite input sequence will be completely consumed after a finite amount of state to state transitions; any further read from memory triggers a bound check, thus terminating the filter.

V. Experimental evaluation

In order to validate the FSA-based packet filtering approach we have run a series of comparisons with a set of other techniques which we believe representative of the current state of the art. Besides FSAs, we have considered BPF which, regardless of its age, is still one of the most common approaches to packet filtering; in order to avoid interpretation overheads we have used the JIT version described in [10]. BPF+ [5] has been selected as it is representative of CFG-based techniques and employs a set of optimization algorithms that allows it to exploit a wider set of opportunities than other techniques regarded as the state of the art, such as PathFinder. BPF+ comes natively with an UltraSparc back-end but it has been modified to generate C code in order to make it compatible with the test platform, similarly to what happens with FSA-based filters. Since the C code for BPF+ filters is compiled with optimizations enabled, it is possible that further optimizations are introduced by the compiler.

NetVM-based filters run on a virtual machine that is explicitely targeted towards packet processing applications. They have been included because they are based on NetPDL protocol descriptions, therefore achieving a level of expressiveness very similar to our approach, especially because NetPDL descriptions can be shared. At the time of the tests being run, the NetVM JIT compiler did not provide any facility to enforce safety, neither in terms of termination nor in terms of memory access correctness. Finally, in order to provide a rough estimate of how synthetic filters compare to hand-written code, in some tests we have included filters hand-coded in C, compiled with the same optimizing compiler used for FSA-based filters. These test programs do not take safety issues into consideration.

It should be noted that each of these techniques has its own filtering language and protocol database, so there are unavoidable differences in capabilities and expressiveness. While the filtering rules were made as similar as possible, whenever meaningful we have decided to let the more modern techniques (FSA-based and NetVM-based) handle advanced features such as multiple levels of encapsulation and full decoding of IPv6, even if BPF+ and BPF cannot.

To ensure significancy, the test filters were run independently in a test bench that measures clock cycles with the RDTSC instruction and uses the `gettimeofday` POSIX system call for longer time 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 of 4 GiB of RAM, running an 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. All tests were performed with hot disk and processor caches.

[5]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`

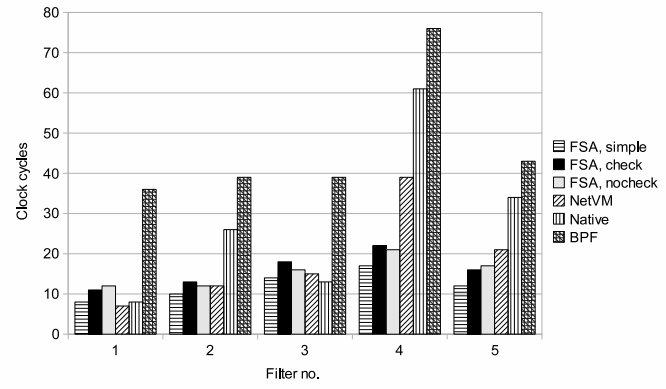


Fig. 6. Code quality evaluation

A. Worst-case filter performance

The first test series aims to evaluate the emitted code quality in terms of clock cycles taken to execute simple filters in the worst case. In order to provide a baseline value to filter complexity in simple cases, we have generated the filters reported in table I with BPF, the NetVM filter generator, by hand and using our generator. 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 was reduced to contain only the complete descriptions of the protocols involved.

For each filter an ad-hoc packet was crafted to trigger the worst-case code path (among those leading to success) to be executed. The test packets did not contain multiple levels of encapsulation, because they cannot be handled by BPF and hand-written C filters. BPF and NetVM measurements refer only to the proper filtering code while the cost of the respective VM frameworks has been excluded.

The results are reported in fig. 6. The two columns "FSA, check" and "FSA, nocheck" refer to FSA-based filters compiled with the aforementioned NetPDL descriptions and bound checks enabled or disabled, respectively. In order to level the field the additional data series "FSA, simple" was generated with a NetPDL database where some protocol features were disabled, to matches more closely the capabilities of the remaining filtering techniques. BPF+ was excluded from this test because no JIT emitter for the x86 platform was available and it would have been hard to separate optimizations introduced by the C compiler from optimizations performed by the filter generator itself.

As it can be seen from the chart, the code quality of FSA-based programs is similar or better than other approaches for filters of different complexity. The best results are achieved in test cases 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. Test cases 1, 2 and 3 show that the FSA-based technique provides low overhead for low-complexity filters as well, even if the protocol database used contains conditions that the other approaches do not handle; when considering the "simple" database the results for FSA-based filters are better than or equal to the other approaches considered in any case.

A second result obtained from benchmark results is that safety checks introduce very low run-time overhead; this is fully justified by the fact that safety check amounted to a single comparison at the beginning of the filter which can be correctly predicted by the CPU, therefore negating most adverse effects. It can be noted that in test cases 1 and 5 enabling safety checks actually lowers filter execution time. This apparently strange behaviour is retained even when tests are repeated 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.

B. TCP session filtering scalability

The filtering techniques were evaluated in more realistic conditions during the second test series, which was designed to highlight 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 captured packet trace; packets were filtered accordingly. The number of recognized TCP sessions was increased from 1 to 128 and the time required to process the aforementioned trace was measured after the disk cache provided by the Linux operating system was preloaded with the packet trace.

The results are presented in fig. 8, which reports the measured average frame rate normalized to the running time of the fastest filter (1 session, FSA-based). The results for NetVM filters are not reported, as they are made less meaningful because of the relatively high overhead introduced by the virtual machine. Since in this case we measure the observed real-world performance of an application performing packet filtering using different filtering techniques, artificially removing the time spent in the framework would result in less relevant measurements. C filters were not considered for this test as well, as it is quite cumbersome to write by hand optimized programs that recognize large numbers of TCP sessions and it is quite unlikely that this operation would be perfomed manually in any case.

The NetPDL database used for this test is represented as an encapsulation graph in fig. 7. The graph is larger than what can be examined using BPF-derived techniques and allows FSA-based filters to recognize TCP sessions even if the IP header is encapsulated. This causes some protocols that are not directly involved in the filter statements (e.g. VLAN and IPv6) to be present in the executable code because their traversal

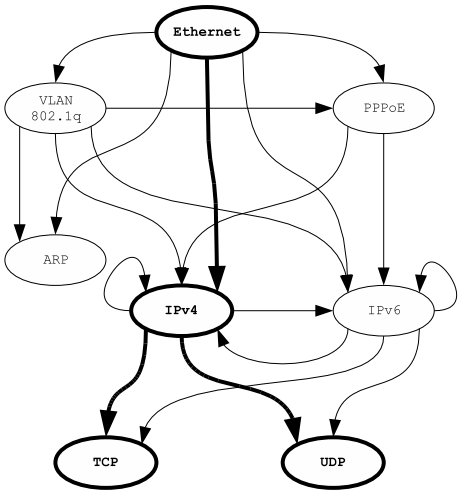


Fig. 7. Protocol encapsulation graph

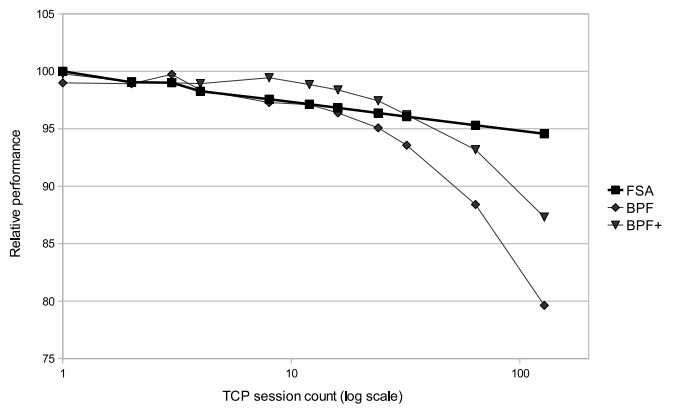


Fig. 8. TCP session filtering performance (log plot)

could be required to reach IP and TCP headers; in turn, this causes more operations to be performed at run-time than the amount required by BPF and BPF+ that do not take tunneling into account. The difference is small but accounts nevertheless introduces some overhead.

FSA-based filters are shown to scale significantly better than the other techniques with increasing session counts: while all the other approaches scale linearly, FSA-based filters follow a logarithmic curve, as expected from the theoretical considerations reported in section IV-D. The absolute frame rate results are good as well, since even in complex cases it is possibile, on the test machine, to filter packets at roughly 150% the rate required for gigabit Ethernet. It is worth noting that the results obtained for low session counts are made somewhat less relevant by the time spent in ancillary tasks such as fetching packets from the trace. Moreover, the CPU of the test machine is probably able to correctly predict a large number of the bounds checks that are present at each memory access both in BPF and BPF+ code, as all packets in the trace are well-formed. Running the same test with a different processor with less advanced branch prediction capabilities would probably yield worse results for BPF and BPF+ where each access is checked. Accurate branch prediction is not as important in FSA-based filters, as they execute only 1 bound check in most cases.

C. Memory consumption and potential state-space explosion

A very felt problem with FSAs, and DFAs in particular, is the potential exponential explosion in the number of states experienced when transforming a NFA into a DFA. This issue is, in the general case, unavoidable and comes from the intrinstic inability of FSAs (especially if deterministic) to cope with certain pattern sets [28] that unfortunately are frequently encountered in real world situations such as intrusion detection systems. A sample pattern set that triggers this behaviour is reported in [29] and consists in multiple patterns of the form "$.*S_i.*S_i'$" (with S_i and S_i' being different strings) that are to be disjunctively matched. In general, a NFA of n states can lead to a DFA of O(2^n) states upon determinization; the DFA minimisation algorithm improves filter memory consumption but does not prevent by itself state-space explosion.

These considerations make it necessary to investigate whether similar issues arise in our context by measuring the memory used by FSA-based filters and its trend with increasing filter complexity. We have measured memory occupation in filters that recognize increasing numbers of TCP sessions. Since no memory is allocated at run-time, except for a very small and fixed stack space, we performed binary code size (which is broadly proportional to state count) measurements with the POSIX `nm` command. This test includes FSA-based and BPF+ filters because they are compiled to object code in a similar fashion. For FSA-based filters we have used a reduced protocol database that mimics the one supported by BPF+, but including all protocol features and all the relevant encapsulation relationships as shown in bold in fig. 7.

The results, presented in fig. 9, show that the memory consumption is broadly linear in the number of filtered TCP sessions and no state explosion occurs for this work set. Moreover, the absolute code size is reasonably small, so even big filters can fit into modern processor caches. FSA-based filters occupy rougly twice the space required for BPF+ filters. 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. If a further reduction in size is desired, a number of different automata compression techniques (orthogonal to our approach) is described in literature [30].

While the performed test provides only empyrical evidence, and it is always possible to design protocols and filter rules to cause a state space explosion, 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, packets always start with a known protocol and follow a known field sequence, so no packet filters start with (or contain) "$.*$" patterns, therefore excluding the chance for the aforementioned case.

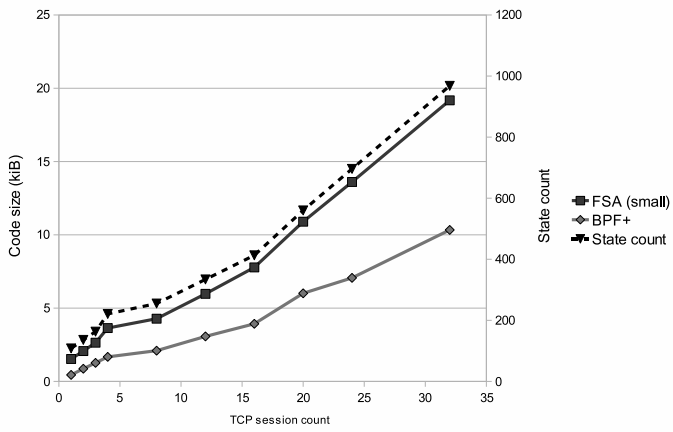


Fig. 9. Memory occupation of TCP session filters

VI. Conclusions and future works

We have designed, prototyped and evaluated a packet filter generation technique based on Finite-State Automata, aimed at obtaining good performance and flexibility while preserving all the traditional safety requirements, both in terms of memory access correctness and termination. The FSA model is particularly valuable for the task at hand because it is powerful enough to express any possible stateless packet filter (even with loops) while at the same time providing a robust mathematical framework with well-defined properties that can be exploited to enforce safety. An interesting property of the FSA model is that, by construction, each packet field is examined at most once; this property always holds and greatly limits the amount of redundancy that can be present in filter programs. Moreover, it does not require any effort to be achieved, in contrast with the large number of optimization algorithms that are traditionally employed and cannot, by their own nature, provide any hard guarantee about the resulting code quality.

There are known algorithms to perform boolean operations between FSAs: this provides a well-defined way to compose filters, an operation that is traditionally perfomed either by running them in sequence or resorting to heuristics that might fail in complex or unforeseen circumstances. FSAs are closed under boolean operations, so even the result of multiple composition operations is still optimized with regard to repeated evaluations of the same packet field.

Finite-state automata are also very well-known and understood and have straightforward and efficient software and hardware implementations: besides being a non-ambiguous semantic model for a filter, they also double as an effective and useful guideline for code emission.

In order to prove this technique to be viable and practical, we have developed a filter generator that creates the filter DFAs from an external protocol database (thus decoupling protocol description from filter generation) and user-specified filtering rules. The DFAs can then be used as they are by existing hardware or software engines, or can be translated into C code by the back-end, which also performs some transformations in order to improve performance. The main optimizations consist in adapting the operations to be performed to the underlying machine word size, and carefully placing a small number of run-time bounds checks in order to enforce filter termination and memory access safety. Bounds checks are generated already fully aggregated, so their number is very low, both in absolute terms (they are generated only on loops or optional parts in the protocol description or in the encapsulation graph) and along any execution path (only one is needed in many cases).

We have evaluated the run-time performance and memory occupation of FSA-based packet filters by comparing them to other approaches both on synthetic benchmarks, by determining the worst-case run-time over a set of simple filters, and on a real-world test, consisting in filtering an increasing number of TCP sessions from a captured packet trace. The resulting programs exceed the dissection capabilities of most other approaches in terms of recognizable fields and tunneling; FSA-based filters are not affected by restrictions, such as forbidding cycles in the filter program, traditionally required in order to enforce termination.

Even if more complex and capable, FSA-based filters are shown to be of comparable performance to other modern approaches such as BPF+ on simple filters; moreover, they are shown to scale better with increasing filter complexity because their regular structure enables a code emission strategy that focuses on improving performance instead of eliminating sources of overhead introduced by the compilation process itself, as it is often the case with traditional compilers. The measured overhead of run-time safety checks is very small and was shown not to cause any significant performance penalties.

Overall, the FSA 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 FSAs and, more specifically, DFAs, is the occurrence of a space-state explosion. This problem is a limiting factor for DFA adoption in pattern-based detectors such as intrusion detection systems; even if it can be triggered with our generator as well, we believe it unlikely to happen with a realistic protocol and common use cases, because of protocol header and filter structure. A specific case that triggers a space state explosion is encountered when filters compare 2 packet fields against each other; while theoretically supported, this requires $O(2^N)$ states for N-bit fields. Similar limitations are shared by other generators, albeit for different reasons. Experimental results show that, when filtering increasing numbers of TCP sessions, memory occupation grows roughly linearly with the numer of filtering rules.

The presented technique can be easily extended to support packet demultiplexing in addition to packet filtering. This is partially supported by our current generator prototype, in that it labels final states with identifiers of the filtering rules that matched and correctly propagates these labels across automata manipulation algorithms. Full support would require handling

dynamic automata creation and code generation, tasks that will be the object of future studies. Another possible extension to our approach consists in enabling interactions (look-ups, updates) with stateful constructs such as session tables, required for higher-layer filtering and traffic classification.

The FSA-based filter generation techniques improves the current state of the art by uniting most of the desirable properties required for packet filters by providing processing speed, reasonable memory consumption, flexibility in specifying the protocol formats and filtering rules, effective filter composition and low run-time overhead for enforcement of both termination and memory access safety. The development of the filter generator and the test results support our claims by showing that the approach presented is viable.

REFERENCES

[1] R. T. Braden, "A pseudo-machine for packet monitoring and statistics," in *SIGCOMM '88: Symposium proceedings on Communications architectures and protocols*. New York, NY, USA: ACM, 1988, pp. 200–209.

[2] A. M. Turing, "On computable numbers, with an application to the entscheidungsproblem," *Proceedings of the London Mathematical Society*, vol. Series 2, 42, pp. 230–265, 1936.

[3] 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.

[4] 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. [Online]. Available: citeseer.ist.psu.edu/bailey94pathfinder.html

[5] O. Morandi, F. Risso, M. Baldi, and A. Baldini, "Enabling flexible packet filtering through dynamic code generation," *Proceedings of the IEEE International Conference on Communications (ICC 2008) - Advances in Networks and Internet Symposium*, May 2008.

[6] S. Ioannidis and K. G. Anagnostakis, "xPF: Packet filtering for low-cost network monitoring," in *Proceedings of the IEEE Workshop on High-Performance Switching and Routing (HPSR)*, 2002, pp. 121–126.

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

[8] Z. Wu, M. Xie, and H. Wang, "Swift: a fast dynamic packet filter," in *NSDI'08: Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*. Berkeley, CA, USA: USENIX Association, 2008, pp. 279–292.

[9] D. R. Engler and M. F. Kaashoek, "DPF: fast, flexible message demultiplexing using dynamic code generation," in *SIGCOMM '96: Conference proceedings on Applications, technologies, architectures, and protocols for computer communications*. New York, NY, USA: ACM, 1996, pp. 53–59.

[10] L. Degioanni, M. Baldi, F. Risso, and G. Varenni, "Profiling and optimization of software-based network-analysis applications," in *SBAC-PAD '03: Proceedings of the 15th Symposium on Computer Architecture and High Performance Computing*. Washington, DC, USA: IEEE Computer Society, 2003, p. 226.

[11] 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.

[12] 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," *Telecommunications, 2005. ConTEL 2005. Proceedings of the 8th International Conference on Telecommunications*, vol. 1, pp. 163–168, June 15-17, 2005.

[13] M. Jayaram, R. Cytron, D. Schmidt, and G. Varghese, "Efficient demultiplexing of network packets by automatic parsing," 1994. [Online]. Available: citeseer.ist.psu.edu/jayaram95efficient.html

[14] H. Bos, W. D. Bruijn, M. Cristea, T. Nguyen, and G. Portokalidis, "FFPF: Fairly fast packet filters," in *Proceedings of OSDI*, 2004, pp. 347–363.

[15] 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.

[16] 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.

[17] 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, NY, USA: ACM, 1994, pp. 159–170.

[18] R. Gupta, D. A. Berson, and J. Z. Fang, "Path profile guided partial redundancy elimination using speculation," in *ICCL '98: Proceedings of the 1998 International Conference on Computer Languages*. Washington, DC, USA: IEEE Computer Society, 1998, p. 230.

[19] V. Laurikari, "Efficient submatch addressing for regular expressions," Master's thesis, Helsinki University of Technology, 2001.

[20] M. Becchi and P. Crowley, "Efficient regular expression evaluation: theory to practice," in *ANCS '08: Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*. New York, NY, USA: ACM, 2008, pp. 50–59.

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

[22] A. Korobeynikov, "Improving switch lowering for the LLVM compiler system," in *Proceedings of the 2007 Spring Young Researchers Colloquium on Software Engineering (SYRCoSE2007)*, May 2007.

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

[24] T. T. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to algorithms*. Cambridge, MA, USA: MIT Press, 1990.

[25] 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.

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

[27] T. Würthinger, C. Wimmer, and H. Mössenböck, "Array bounds check elimination for the java hotspot™client compiler," in *PPPJ '07: Proceedings of the 5th international symposium on Principles and practice of programming in Java*. New York, NY, USA: ACM, 2007, pp. 125–133.

[28] S. Kumar, B. Chandrasekaran, J. Turner, and G. Varghese, "Curing regular expressions matching algorithms from insomnia, amnesia, and acalculia," in *ANCS '07: Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems*. New York, NY, USA: ACM, 2007, pp. 155–164.

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

[30] D. Ficara, S. Giordano, and G. Procissi, "An improved DFA for fast regular expression matching," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 5, pp. 29–40, 2008.