Ruler: high-speed traffic classification and rewriting using regular expressions

– Technical Report –

Kees van Reeuwijk and Herbert Bos
Vrije Universiteit Amsterdam
Email: {reeuwijk,herbertb}@cs.vu.nl

***Abstract*— We describe Ruler, a flexible language for network traffic inspection and rewriting. Ruler was designed to support anonymisation at high link rates. As anonymisation requires a trade-off between privacy and usefulness of the anonymised data, flexibility is essential. For this purpose, Ruler allows matching of arbitrary traffic patterns by means of regular expressions, and construction of arbitrary output packets from fragments of the input packets. However, we show that Ruler is sufficiently general to be useful in other application domains like intrusion detection. For pattern matching, Ruler uses Deterministic Finite Automata (DFAs) which can be implemented in virtually any execution environment, including embedded hardware. Rewriting data based on regular expressions requires administration of intermediate positions in patterns, which is beyond standard DFA construction techniques. We therefore used *tagged* DFAs where tags are markers in partially matched patterns. Ruler is capable of handling multi-gigabit link rates on a standard processor for a large range of expressions.**

I. Introduction

Ruler is a language to match and rewrite network data. It was originally intended for traffic anonymisation in online network monitors and designed for both flexibility and speed. However, Ruler is useful in other application domains as well.

Flexibility in anonymisation is important, because a delicate balance must be struck between privacy and security concerns on the one hand, and the information needs of the network monitoring application on the other [1]. Speed is equally important, as Ruler is intended for online processing. Offline trace anonymisation is mainly useful for providing the research community with safe access to real traffic for testing purposes and post-mortem analysis. Online rewriting allows real-time monitoring of the network, which is crucial for malware detection, trend spotting, etc.

In both speed and flexibility Ruler outperforms related approaches like [2], [3]. For instance, unlike most other projects, Ruler supports both filter-in (specifying what traffic remains unmodified), and filter-out semantics (specifying what traffic should be changed or dropped). The approaches may even be combined. For classification at gigabit rates current approaches are often limited to header fields. If full payload scanning at these rates is supported at all, the common approach is to restrict the scan to variations of exactly matching single strings [4], [5], often by means of specialised hardware, like CAMs [6]. For realistic classification and rewriting, more generic matching such as provided by regular expressions constitutes a huge improvement over existing methods. To the best of our knowledge, we are the first to offer such functionality at gigabit rates on commodity hardware.

Ruler relies on deterministic finite automata to provide efficient filter implementations, permitting us to process traffic at link speeds well above a gigabit/s on a common CPU and a DAG network card. The language is simple but powerful and revolves around well-known regular expressions. Moreover, rewriting/anonymisation functions in Ruler are not cast in stone, as the language is extensible with external functions. The extensibility permits a user to plug in a novel prefix-preserving anonymisation function, say, without waiting for it to be incorporated in the language. The Ruler distribution comes with a library of field anonymisation functions, so as to enable the use of functionality like advanced hashing without providing it in the language.

Moreover, Ruler's flexibility in packet filtering and rewriting makes it a useful tool in a wider range of network processing applications. As an example we will show how Ruler can do intrusion detection using Snort rules [7].

In this paper, we present the Ruler language, as well as its implementation. Our compiler translates Ruler filter descriptions into efficient code and is capable of generating code for different back-ends, including user-space C programs, kernel modules, and assembly language for embedded network processors. The code interfaces with a support framework for accessing the network packets. We currently support the well-known packet capture library pcap [8], the monitoring API (MAPI [9]), the DAG card library [10], and a home-grown framework for network processors. Our compiler uses *templates* [11] to generate both the code for the support environment and the packet matching itself. Therefore, supporting new network infrastructures, application areas, or programming languages all require little additional effort.

In summary, the contributions of this paper are the following. First, we have designed a *high-level, flexible* rule-based language that allows administrators to specify in a friendly manner how network traffic (packets and/or streams) should be rewritten. A rule in this language consists of a regular expression pattern that is matched against the traffic and a rewrite action that is executed when there is a match. Second, Ruler is *fast*: we compile the code to an efficient DFA that checks all rules simultaneously, by processing traffic one byte

at a time. As a result, we can process multi-gigabit link rates on commodity hardware. Third, Ruler's DFAs are *tagged*, which permits the rewrite action to refer to parts of the matching pattern. To our knowledge, using tagged DFAs is unique in this field. Since it enables efficient packet rewriting, we consider this an important contribution. Fourth, the language and its implementation contain many *network-oriented optimisations* that both speed up processing and simplify the administrator's job. In addition, we show that our solution is practical and can be used for anonymisation as well as other classification and rewriting applications (for intrusion detection, NAT, etc.).

In Section II we describe the Ruler language. In Section III we describe the compilation process. Section IV discusses implementations and Section V describes a case study in intrusion detection. In Section VI we evaluate Ruler's performance, in Section VII we describe related work, and in Section VIII we draw conclusions. In the appendix to this paper we show the exact ruler filters used for most of the benchmarks. Ruler is available from `gforge.cs.vu.nl/projects/ruler`.

II. The Ruler language

As explained, Ruler is a language to match and rewrite network data. A *filter* is the definition of such a match-and-rewrite operation. A filter consists of a list of match-and-rewrite specifications, called *rules*. A rule consists of a *match* pattern and a *result* pattern. The match pattern specifies the layout and specific values of packets. Since Ruler is designed for network processing, we call a unit of data a *packet* in this paper. However, Ruler works on any sequence of bytes. For example, in Section V we will show an application where the unit of data is an entire TCP stream. Indeed, in a suitable context Ruler may work on a packet, a line of text, a database record, a TCP stream, or a file.

The rule

```
byte#26 192 168 1 1 * => accept;
```

matches all packets that start with 26 arbitrary bytes, followed by four bytes with specific values, followed by any number of bytes. The result pattern of this rule is the keyword `accept`, meaning that the packet should be output without modification. Given a stream of IPv4 on Ethernet frames, this rule accepts all packets that contain the source address 192.168.1.1. We will show later how to make this rule more readable.

Similarly, the keyword `reject` means that the packet should not be output, so

```
byte#26 192 168 1 2 * => reject;
```

rejects all packets that contain the source address 192.168.1.2.

Finally, the result pattern can specify a new packet:

```
h:byte#26 192 168 1 1 t:* => h 0 0 0 0 t;
```

copies the patterns labelled `h` and `t` to the output, but replaces the four constant bytes with zero bytes. Here `h` and `t` are *labels* for parts of the match pattern. Given a stream of IPv4 Ethernet packets, this rule anonymises the source address of IPv4 packet headers.

If no length is specified, a constant is 1 byte or 1 bit. Other lengths must be specified explicitly with a `~` expression, e.g. `0~3` is three bytes. Byte constants represent big-endian patterns, so `258~2` is equivalent to the sequence `1 2`. A sequence of characters with a double quote character (‘`"`’) at each end matches the sequence of ASCII codes (bytes) of the characters. Thus, the pattern `"abc"` is equivalent to the pattern `97 98 99`.

The `*` pattern denotes an arbitrary number of arbitrary characters. For efficiency, the `*` matches the *shortest* possible span of bytes. In contrast, many traditional regular expression languages use the *longest* match. However, this requires an exhaustive search of all possible matches.

A sequence of patterns can be grouped by surrounding it with round brackets. This allows a more complicated sequence to be labelled. Such a labelled group of patterns may itself also contain labels. One can refer to the inner labels by prefixing them with the label of the sequence and a dot (‘`.`’). This is called a *qualified* label. For example, in the pattern:

```
source:(network:byte#2 machine:byte#2)
```

the label `source` refers to the entire sequence of 4 bytes. The first two bytes in the sequence have the qualified label `source.network`.

A. Bit patterns

Ruler supports *bit* patterns which are denoted by braces ‘{’ and ‘}’. For example, the rule:

```
h:(byte#14 {4~4 bit#4}) {bit#3 r:bit#5} t:*
  =>
h {0~3 r} t;
```

matches all packets with the value 4 in the highest four bits of the fifteenth byte. The lowest four bits of byte fifteen may have any value. The sixteenth byte may have any value, but is conceptually split in two parts: three unlabelled bits and five bits labelled ‘`r`’. The pattern is followed by a tail ‘`t`’ consisting of an arbitrary number of unspecified bytes. In case of a match, the rule zeroes the highest three bits in the sixteenth byte of a packet. Given a stream of Ethernet packets, this rule matches IP packets with the IP version number 4, and zeroes the precedence field in the IPv4 header.

B. Pattern definitions

Frequently occurring patterns can be defined separately, so that their layout can be reused. For example, the following definition describes the layout of the UDP message header:

```
pattern UDP: (src:byte#2 dst:byte#2
  len:byte#2 csum:byte#2)
```

Such a pattern can be used in matching and result patterns. For example:

```
h:UDP t:* => h.src 0~2 h.len 0~2 t
```

Note the references to fields of `UDP` in the result pattern (`h.src` and `h.len`).

C. Pattern substitution

The `with` construct replaces parts of a pattern with a newly specified value. This should not be confused with packet rewriting: the `with` construct is intended to create specialised

versions of patterns. For example, using the `UDP` definition of the previous section

```
UDP with [ src=1234~2 ]
```

is equivalent with the pattern:

```
(src:1234~2 dst:byte#2
  len:byte#2 csum:byte#2)
```

This matches all UDP packets coming from port `1234`.

D. Result patterns

The pattern at the right-hand side of a rewrite rule must be fully known, so it may not contain ⋆ patterns or non-constant bit or byte patterns. The right-hand side may also contain the application of other filters. For example:

```
pattern address: (netwrk:byte#2 host:byte#2)

filter zap_host : 4 :
    a:address => a with [host=0~2];

filter clear_hosts
    h:byte#26 src:address dst:address t:⋆
        =>
    h @zap_host(src) @zap_host(dst) t;
```

The top-level filter '`clear_hosts`' matches a header '`h`' of 26 unspecified bytes (presumably an Ethernet header and the first 12 bytes of the IP header), followed by a source and destination address and the remainder of the packet. The result pattern applies the filter '`zap_host`' filter to both addresses. The filter `zap_host` matches a four-byte address consisting of a network part and a host part. It rewrites this address to a form that leaves the network part untouched, and zeroes the host part.

The *@foo()* construct also permits us to call externally defined functions. For instance, if a new prefix-preserving hash function is implemented in C, it may be used from within Ruler using the `@` notation. In this way, we added an extensible library of field-anonymisation functions to Ruler.

E. From rules to filters

A *filter* consists of a list of rules. The rules are matched in the order that they are specified; the first matching rule determines the result. Any packet that doesn't match a rule is rejected. For example, the filter:

```
filter get_spam
    ⋆ "'warez'" t:⋆ => accept;
    h:⋆ "warez" t:⋆ => h "xxx" t;
```

replaces the first match of the string `warez` with the string `xxx` in the output packet, but leaves unchanged all packets that contain `warez` surrounded with single quotes. All other packets are rejected.

III. Constructing the state machine

To match packets against a Ruler pattern we generate a finite state machine that examines the bytes of a packet one by one, and changes state depending on the value of these bytes. The advantage of this approach is that the matching process is very time-efficient, which is essential for this application.

Ruler match patterns are regular expressions. Compiling regular expressions is a well-studied problem [12], although

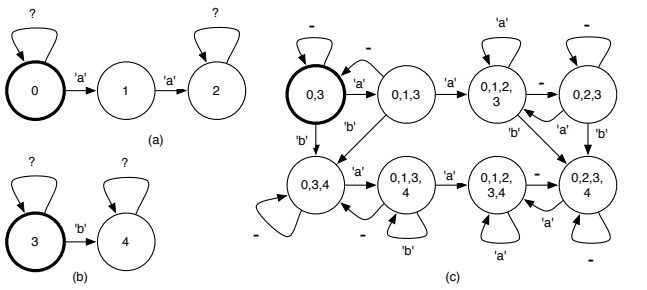

Fig. 1. NFAs to match (a) '⋆ "aa" ⋆', (b) '⋆ "b" ⋆', and (c) a DFA constructed from these machines using the subset algorithm. A '?' transition is taken on any input character; a '-' transition is taken if no other transition is taken.

we shall see that Ruler requires a generalisation of the traditional methods. The standard approach is to first generate a Non-deterministic Finite Automaton (NFA) from the regular expressions. Although more elaborate translation schemes may be used [13], in a straightforward translation an NFA state simply indicates that a particular point in the match pattern has been reached. For example, in Fig. 1a state 0 indicates that no `aa` sequence has been seen, state 1 indicates that the first character of the `aa` sequence has been seen, and state 2 indicates that the entire `aa` sequence has been seen. Since seeing a single `a` does not guarantee that the entire `aa` sequence will be seen, the NFA also keeps state 0 active to cover the case where the full match fails.

The NFA is converted to a Deterministic Finite Automaton (DFA) using the *subset* algorithm. The start state of the DFA is the set of all start states of the NFAs. For each set of NFA states and for every possible transition, the set of destination NFA states of a transition is collected into a set. Each distinct set is a DFA state.

Fig. 1c shows the resulting states for this algorithm on the two NFAs in Fig. 1a and 1b. For example, state 0,3 represents the start state of the DFA, where only states 0 and 3 of the two NFAs are active. A transition on `a` leads to state 0,1,3.

As explained, the rules in a filter have decreasing priority. To determine which pattern, and hence which rule, matches, we can run the engine for all bytes in the input packet, and then look at the state of the underlying NFAs. The highest-priority NFA that is in its end state determines the matching rule. We ensure that there is always such a rule by adding an implicit `⋆ => reject;` rule as lowest-priority rule.

In many cases it is possible to stop the matching process before all bytes in the packet have been processed. For example, if we assume that the NFA of Fig. 1a has the highest priority, in Fig. 1c we can stop once we reach a DFA state that contains NFA state 2, because the result will be the same for any remaining input.

The subset algorithm does not produce the most compact DFA. For example, in Fig. 1c states 0,1,2,3 and 0,2,3 are equivalent, and may as well be merged. We use a standard algorithm [14] to construct a DFA with the smallest number of states. Although many inefficiencies in DFA construction

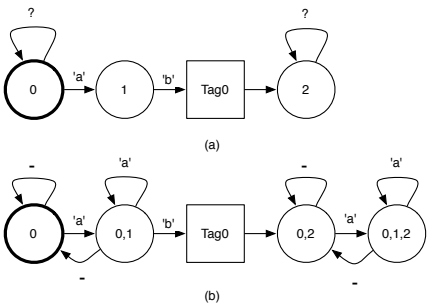

Fig. 2. (a) TNFA to match '⋆ "ab" ⋆', and determine the position of the string "ab", and (b) TDFA constructed from the TNFA.

will be undone by this minimisation algorithm, intermediate DFAs may well be impractically large. We therefore also apply a number of optimisations to construct a more compact NFA.

A. Tagged NFAs and DFAs

Ruler introduces one important complication: it requires the positions of intermediates in the matched text. For example, the filter:

```
filter example
  x:byte#4 y:* "ab" z:* => y z;
```

requires that the start and end position of the labelled patterns `y` and `z` are known, so that these spans of text can be copied to the output. Some of these positions, such as the end of `z` and the start of `y`, are known at compile time, but the other two are determined by the input.

To solve this potentially complex problem, we use theory developed by Laurikari [15] which describes a generalisation of the DFA construction process by introducing *Tagged* NFAs and DFAs (TNFAs and TDFAs). Conceptually a TNFA is similar to a 'normal' NFA, but special states are introduced to record positions. The translation from NFA to DFA is generalised to handle these states. Figure 2 shows the TNFA and TDFA for the `example` filter. The tag state, represented as a rectangle, records the current position in the packet in a variable (`Tag0`), and then continues to the next state.

B. Ruler support for network traffic

The Ruler DFA is further optimised for traffic processing. Network packets often contain fixed-length stretches of bytes that need not be inspected at all, such as header fields. Instead of having the DFA go through the motions for these bytes, we support 'jump' states that skip past them, saving cycles. Also, we have provisions for content-dependent field lengths, such as IP headers, whose length is defined in the header itself. In other words, we may instruct Ruler to jump to an offset that is determined by the IP header length field to start processing TCP headers directly.

C. Limitations

A potential limitation of Ruler is the size of the DFA. All state in the matching process must be represented as separate states, potentially leading to large DFAs. The DFA grows with the number and complexity of rules. For anonymisation, this is not likely to be a problem. While in a scenario with thousands of rules, the DFA may become unwieldy, many techniques for DFA-size optimisation exist (e.g., a complex DFA may be split into multiple smaller ones).

IV. Implementations

Ruler is capable of generating (a) executable DFAs where the DFA itself is embedded in the code and each state is essentially a `switch` statement and each `case` clause jumps to the next state directly, and (b) interpreted (or table-driven) DFAs where the DFA is a table containing all the states and transitions of the DFA; the only active code is a simple interpreter that reads the next input byte, and looks up the next state in memory. Executable DFAs are faster. They compile to native object code and need fewer memory references. On the other hand, when the number of rules grows, the code size tends to be large, which has its own disadvantages. For instance, large programs may not fit in the instruction stores of embedded processors.

A. Code generation targets

Besides the various types of DFA implementation, we have implemented back-ends for various targets. By employing the appropriate template, we are currently able to produce:

- User space C code;
- Linux kernel space C code;
- Assembly for Intel IXP2xxx network processors;
- Verilog (work in progress).

In addition, the compiler templates bind the generated DFAs to various environments for traffic capturing. The current implementation supports the well-known packet capture library pcap [8], the monitoring API (MAPI [9]), the FFPF API [16], the DAG card library [10], and a home-grown framework for network processors.

B. Embedding Ruler

We have not yet mentioned what should happen to the rewritten data or packets. Options are to store mangled packets in a trace file, feed them to a monitoring application on the host, or to put modified data back on the wire for retransmission. In addition, certain packets may be dropped, while others may pass untouched. Given the plethora of applications in which the language could be used, Ruler does not prescribe any way of handling the data or packets that it has processed. Instead, the decision of what to do with the data is left to the environment in which the Ruler filter is embedded.

To facilitate the process, we provide optional accept values. A rule may be given a suffix of the form '`-> accept` *n*'. The parameter '*n*' can subsequently be used for classification. For instance, in the intrusion detection system (IDS) that we implemented on a programmable network card (discussed in the next section), we use three types of accept values, in addition to '`reject`'. These values correspond to the following actions: (1) forward packets on the wire, (2) forward packets across the PCI bus to the host, and (3) forward packets across the PCI bus onto the network link.

V. Case study: Ruler for IDS/IPS

To illustrate that Ruler is applicable to domains other than anonymisation, we describe *SafeCard*, an intrusion detection/prevention system running entirely on a programmable network processor, embedded on a Radisys ENP2611 network card. The 600 MHz Intel IXP2400 network processor contains on-chip a general purpose XScale control processor and 8 specialised stream processors called *microengines*. Each microengine contains a small instruction store. Besides the network processor, the board contains large amounts of relatively slow SRAM and DRAM memory.

Ruler microengines. We dedicate one microengine each to packet reception and transmission, and another one to TCP reassembly. The remaining five microengines are used for Ruler. Rather than rewriting packets, the IPS drops packets containing malicious content. The open TCP flows are collected in a pool, and the microengines take their work from this pool. While a TCP flow may be handled by different microengines during its lifetime, only one microengine is processing it at any one time. The XScale is used for higher-level protocol-specific intrusion detection.

`Snort2Ruler` compiler. Snort is perhaps the most widely deployed network IDS worldwide [7]. It uses a rule-driven language, which combines the benefits of signature, protocol and anomaly based inspection methods. Some of its rules employ regular expressions, but these are treated as exceptions in user space and the rules are fairly inefficient. Our aim is to build on the enormous rule base available for Snort, while executing more efficiently in hardware. For this purpose, we developed a compiler that automatically translates Snort rules to Ruler. On the five Ruler engines we essentially evaluate a set of well-known Snort rules, translated to Ruler.

Hybrid DFAs. In addition, the Ruler implementation in SafeCard is exploiting locality of reference in the DFA by taking a hybrid approach that combines executable DFAs with interpreted ones. Since the instruction store is so small, it is unlikely that the entire DFA will fit. Instead, we store the hottest states (in practice, the top of the DFA) in the instruction stores as executable code, and the rest as an interpreted DFA in (slow) memory.

The details of SafeCard are beyond the scope of this paper. Interested readers are referred to [17]. In summary, Ruler was used quite successfully and easily kept up with gigabit rates, despite running on fairly outdated hardware.

VI. Evaluation

Even before we look at actual performance figures, it is clear that we can expect large differences in the performance of a Ruler filter depending on the complexity of the filter, the average number of bytes the filter must inspect, and on the actions on the packets it must perform. To properly characterise the performance of Ruler under different circumstances, we examine a number of different aspects that might influence performance: the cost of classifying the packets, the cost of processing the packets once they have been classified, and the characteristics of the traffic we are monitoring.

With packet classification we mean the processing that is needed to determine which action must be performed on the packet[1]. We expect the cost of packet classification to be directly related to the number of bytes in the packet that must be examined. We also expect a complex DFA machine to be slower, due to the more complex code of each state and the poorer cache behaviour.

For our benchmarks we therefore consider three representative cases: packet classification based on the packet header, inspection of the full packet with a simple state machine, and full packet inspection with a complex state machine.

Ruler supports three modes of packet processing with significant differences in performance. In order of increasing cost they are: only accept or reject a packet, in-place modification of a packet, and a general packet rewrite that requires the bytes to be copied to another buffer.

For the user of a Ruler filter it is not interesting how many clock cycles the filter requires, but only whether it can handle the network traffic, no matter what its volume and contents. As it turns out, Ruler is able process all traffic with all filters in our benchmark set. For the evaluation in this paper we delve a little deeper. We instrument the code to determine the number of processor clock cycles spent in the Ruler filter itself, and in the total execution of the filter. From that we compute the percentage of execution time spent in the Ruler filter. Since in practice this percentage can reach more than 95% without packet loss, it is a good indication of the 'headroom' of the system for each filter.

We evaluate the performance of Ruler using two sets of filters. First, filters designed to highlight individual performance aspects of Ruler:

name	**action**	**match on**
aa	accept/reject	Anything
ha	accept/reject	Header
hi	in-place rewrite	Header
hr	full rewrite	Header
pa	accept/reject	Payload
pi	in-place rewrite	Payload
pr	full rewrite	Payload
la	accept/reject	Payload

Second, filters that are representative of realistic Ruler filters:

name	**description**
all	Zero all addresses in all protocol fields of all IP and ARP packets, drop all payload, drop all other traffic.
www	Only pass all TCP traffic to port 80; zero addresses, but keep payload.
snort	Only pass packets that match the Snort 'backdoor' rules.

In the Appendix these filters are described in detail, including actual listings.

[1]This is not the same as determining which rule must be applied: it is sufficient to determine that all remaining possible rules have the same action.

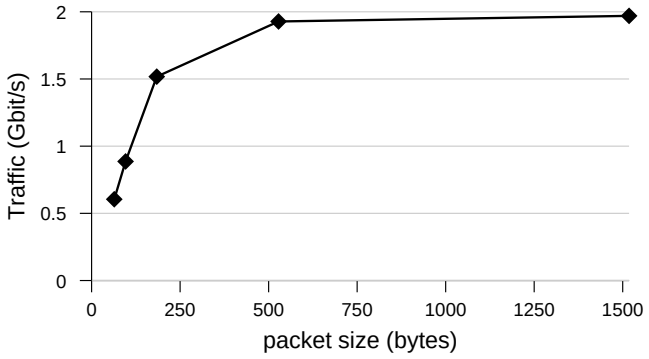

Fig. 3. Generated traffic for different packet sizes.

The state machines associated with these filters vary widely in size. Some of them have only a few states (e.g. the `ha` filter has 22), while the `la` filter has nearly 20,000 states.

We first apply these filters to streams of identical packets that are known to stress the filters. This allows us to determine the performance limitations of Ruler -generated filters. To evaluate the performance of the filters on more realistic network traffic, we also apply the same filters to replays of real traffic.

A. The test system

Our measurement setup contains two clusters of computers with standard gigabit NICs that each generate traffic over a dedicated gigabit link. Each link is fed to a port of an Endace DAG 4.3GE network monitoring card hosted in a dual-processor 1.7GHz Opteron system. The DAG card transfers the packets to the main memory of the host system, where it is handled by a Ruler filter running as a user-space program.

Since we want to evaluate the performance of Ruler under maximum load, we want to saturate both the network links to the network monitoring card. Unfortunately, we are not able to accomplish this for small packet sizes. Figure 3 shows the total amount of traffic on the links for various packet sizes.

B. Synthetic traffic

The traffic that we monitor in this experiment consists of two types of UDP packets; the filters accept one type, and reject the other. The two packet types differ in their UDP destination port numbers, and only the 'reject' packets have the string `REJECT` in their payload. Some Ruler filters use the port number to discriminate between the packet types, while others search for the string `REJECT` in the packet payload. The latter type examines the entire packet for accepted packets, but only the first few bytes of the rejected traffic. We transmit three kinds of traffic, depending on the fraction of accepted ('difficult') traffic in it: 100% accept, 50% accept, and 25%. We also vary the length of the transmitted packets.

Figure 4 shows the results for the various filters for traffic with 100% 'difficult' packets. As expected, filters that process only packet headers are very cheap, whereas inspection of the full packet is much more expensive. For large packets a smaller fraction of the bytes is part of the header, so we expect filters that only examine the header to require less execution time for a stream of larger packets. This effect is visible in the results, but it is masked by the smaller amount of traffic we generate for smaller packet sizes. Rewriting the packet has a small but noticeable impact on the performance of the filter.

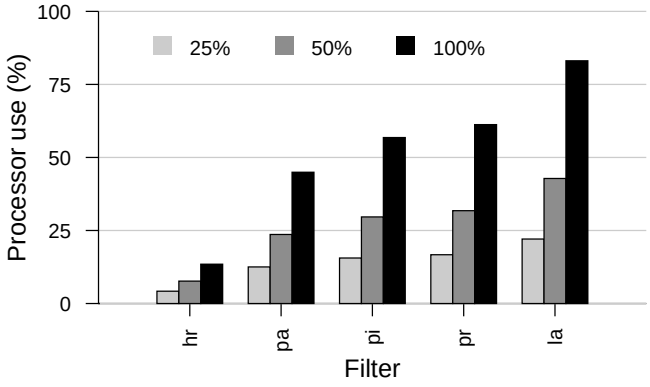

Fig. 5. Processor use for selected benchmark filters for different percentages of 'difficult' packets.

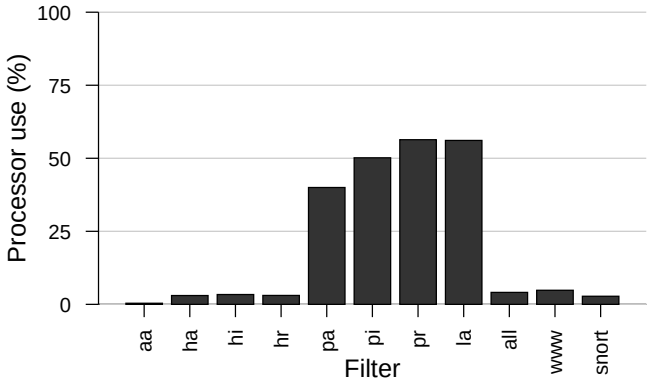

Fig. 6. Processor use for selected benchmark filters for real network traffic.

Figure 5 repeats some of the results of Fig. 4, and compares them with results for traffic with only 50% and 25% difficult packets. As expected, the processor load is proportionally lower for traffic with a lower fraction of difficult packets.

C. Real traffic

Now that we have examined the performance of Ruler on synthetic workloads, it is useful to examine its performance on more realistic network traffic. The most obvious approach would be to monitor a real data link in a real system, but there are some practical problems with this. First of all, real data links are rarely saturated, and network administrators want to stay well away from this state. Another reason we cannot do our measurements on a real data link is privacy.

To avoid these problems we again use the setup described in the previous section, and we again generate our own network traffic. In this case, however, we use a pre-recorded data stream

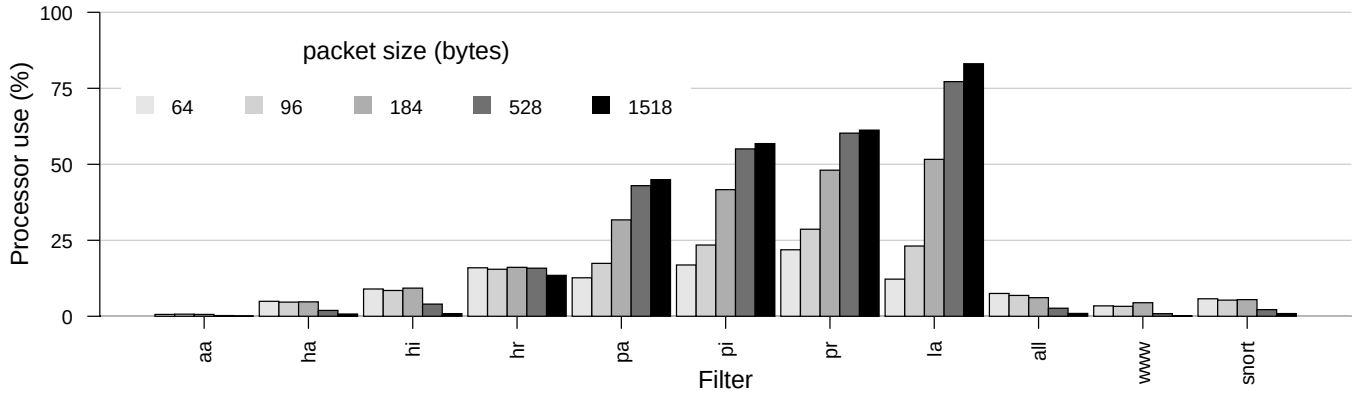

Fig. 4. Processor use for the different benchmark filters and synthetic traffic with various packet sizes. All measurements are for 100% 'difficult' packets.

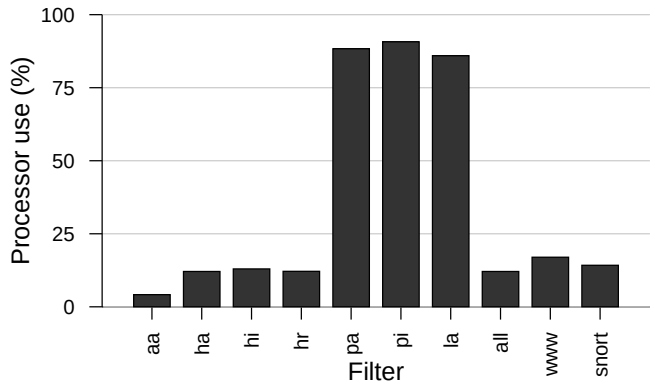

Fig. 7. Processor use for selected benchmark filters for real network traffic and a **table-driven** DFA implementation. Filter `pr` is not shown because the table implementation is too slow to handle the traffic. Figure 6 shows the results for a DFA implemented in C code.

from the 1998 DARPA Intrusion Detection Evaluation Data Set [18], [19]. In particular, we use the trace file for Monday of the first week of training data, a 266Mbyte tcpdump trace file. Since we are still interested in stress-testing the system, we ignore the timing information in the traces, and replay the traffic as fast as possible. The average packet length in the traces is 350 bytes, and our system is able to generate traffic at more than 1.8 Gbit/s.

Figure 6 shows the result for our benchmark filters on this traffic. The results show that our filters are easily able to handle the filtering, even at this high traffic rate. Figure 7 show the same benchmarks for the table-driven implementation described in Sec. IV. Note that the result for the `pr` filter is not shown because this implementation of the filter was too slow to handle all traffic.

VII. Related work

In this section, we compare our work with other anonymisation frameworks and work on high-speed regular expression execution. We do not discuss individual anonymisation functions (e.g., prefix preserving, sequence numbers, etc.), as these functions can always be plugged into a generic framework.

The anonymisation extension to Bro has goals similar to ours, even if their approach is different [3]. However, they mainly focus on the processing of previously captured traces, although slow links may be processed live. In contrast, Ruler is designed for live processing of fast links. This fundamental difference in design focus is visible in many aspects of the two designs. First, in Ruler all traffic that matches a pattern is rewritten according to the result pattern. In contrast, Bro generates events for specific protocol-semantic elements, e.g., the arrival of an SMTP message. At that point, Bro determines whether the packet needs to be rewritten, and how. As a consequence, many events may be triggered that result in no action. Second, Ruler is implemented as a highly-efficient DFA, while Bro scripts are interpreted. Third, by not focusing exclusively on protocol-semantic elements, but rather on patterns across all protocol fields, we exploit some Integrated Layer Processing (ILP), known to improve performance [20], while Bro is hierarchical. Fourth, Bro runs on top of the (slow) user-level pcap library. While Ruler supports pcap, we also interface faster capturing APIs, and have implementations in the kernel and on embedded network processors.

Another well-known anonymisation tool on top of pcap is Mishall's `tcpdpriv` [2]. The tool is somewhat less flexible than Bro's extension and also aims mainly at offline processing of previously captured traces.

Unlike Bro and `tcpdpriv`, Ruler is also amenable to implementation in hardware such as FPGAs. Apart from the performance gain, doing so has the advantage that non-anonymised data never leaves the monitoring hardware, making eavesdropping on un-anonymised data harder.

If we restrict ourselves to packet inspection and filtering, as opposed to packet rewriting, we can identify some additional related work. In-kernel and even on-board packet filtering is fairly common [21] but the expressiveness of the languages is usually limited and, for high-speed implementations, often does not include regular expressions. If full payload scanning at these rates is supported at all, the common approach is to

restrict the scan to variations of string matching [4], [5], often by means of specialised hardware, like CAMs [6].

While regular expression matching is used in IDSs such as Snort [7], it is often used for special cases and not intended for processing the full payload of all traffic streams. Instead, efficient *string* matching like Aho-Corasick [22] is used in the common case. In contrast, in Ruler every pattern is a regular expression. However, for string matching, the Ruler DFA is identical to that of Aho-Corasick and hence has the same performance.

Improving regular expression and DFA implementations is still an active area of research [23], [24]. We have applied several text-book level techniques for improving our DFAs, in addition to generalising the DFAs to tagged DFAs which we believe to be unique in this field.

Since deep packet inspection is a well-structured but demanding application, it is attractive to implement it on specialised hardware, in particular FPGAs. Such systems have been described for fixed strings [25], [26], and regular expressions [27], [28]. Generalisation to tagged regular expressions should not pose any fundamental problems, so Ruler is also suitable for FPGA implementation. Ruler is already capable of generating Verilog for the DFAs, but evaluation, debugging and improvement of the code is ongoing work.

VIII. Conclusions and future work

We have presented Ruler, a flexible language for inspecting and rewriting network data at gigabit line rates by mapping regular expressions to deterministic finite automata (DFAs). Without packet loss and on standard hardware, Ruler is capable of matching complicated input patterns based on regular expressions and of rewriting network data as a function of these input patterns. To support this functionality we extended the DFAs with tags. Ruler's speed, flexibility and tagging exceed those of most existing techniques and make it suitable for a wider range of tasks. We have demonstrated how we applied it to intrusion detection. Current work involves development of a mature backend for FPGAs. In the future, we will extend the language with typing to allow formal reasoning about rewrite rules.

Acknowledgements

This research was funded by the EU FP6 Lobster project. We would like to thank Tomas Hruby for his implementation of Ruler on the IXPs and Willem de Bruijn and Kees Verstoep for commenting on earlier versions of this paper.

References

[1] R. Pang, M. Allman, V. Paxson, and J. Lee, "The devil and packet trace anonymization," *SIGCOMM Comput. Commun. Rev.*, vol. 36, no. 1, pp. 29–38, 2006.

[2] G. Minshall, "Tcpdpriv: Program for eliminating confidential information from traces," http://ita.ee.lbl.gov/html/contrib/tcpdpriv.html.

[3] R. Pang and V. Paxson, "A high-level programming environment for packet trace anonymization and transformation," in *SIGCOMM '03: Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*. New York, NY, USA: ACM Press, 2003, pp. 339–351.

[4] J.-S. Sung, S.-M. Kang, and T.-G. Kwon, "A fast pattern-matching algorithm for network intrusion detection system." in *Networking*, ser. Lecture Notes in Computer Science, F. Boavida, T. Plagemann, B. Stiller, C. Westphal, and E. Monteiro, Eds., vol. 3976. Springer, 2006, pp. 1157–1162.

[5] R.-T. Liu, N.-F. Huang, C.-H. Chen, and C.-N. Kao, "A fast string-matching algorithm for network processor-based intrusion detection system," *Trans. on Embedded Computing Sys.*, vol. 3, no. 3, pp. 614–633, 2004.

[6] F. Yu, R. H. Katz, and T. V. Lakshman, "Gigabit rate packet pattern-matching using TCAM," in *ICNP '04: Proceedings of the Network Protocols, 12th IEEE International Conference on (ICNP'04)*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 174–183.

[7] M. Roesch, "Snort: Lightweight intrusion detection for networks," in *Proceedings of the 1999 USENIX LISA Systems Adminstration Conference*, 1999. [Online]. Available: http://www.snort.org/

[8] V. Jacobson, S. McCanne, and C. Leres, *Pcap - Packet Capture library*, Lawrence Berkely Laboratory, Berkeley, CA, October 1997.

[9] M. Polychronakis, E. Markatos, K. Anagnostakis, and A. Oslebo, "Design of an application programming interface for ip network monitoring," in *IEEE/IFIP Network Operations and Management Symposium*, Seoul, Korea, April 2004.

[10] J. Cleary, S. Donnelly, I. Graham, A. McGregor, and M. Pearson, "Design principles for accurate passive measurement," in *Proceedings of PAM*, Hamilton, New Zealand, Apr. 2000.

[11] C. v. Reeuwijk, "Rapid and robust compiler construction using template-based metacompilation," in *Proc. of the Compiler Construction conference*, Warsaw, Poland, Apr. 2003.

[12] J. Hopcroft and J. Ullman, *Introduction to Automata Theory, Languages, and Compilation*. Addison Wesley, 1979.

[13] A. Brüggemann-Klein, "Regular expressions into finite automata." in *LATIN*, ser. Lecture Notes in Computer Science, I. Simon, Ed., vol. 583. Springer, 1992, pp. 87–98.

[14] J. Hopcroft, "An $n \log n$ algorithm for minimizing states in a finite automaton," *Theory of Machines and Computation*, pp. 189–196, 1971.

[15] V. Laurikari, "NFAs with tagged transitions, their conversion to deterministic automata and application to regular expressions," in *SPIRE*, 2000, pp. 181–187.

[16] H. Bos, W. de Bruijn, M. Cristea, T. Nguyen, and G. Portokalidis, "FFPF: Fairly Fast Packet Filters," in *Proceedings of OSDI'04*, San Francisco, CA, December 2004.

[17] W. de Bruijn, A. Slowinska, K. van Reeuwijk, T. Hruby, L. Xu, and H. Bos, "Safecard: a gigabit IPS on the network card," in *Proceedings of 9th International Symposium on Recent Advances in Intrusion Detection (RAID'06)*, Hamburg, Germany, September 2006.

[18] "1998 DARPA intrusion detection evaluation data set," webpage, www.ll.mit.edu/IST/ideval/data/1998/1998_data_index.html.

[19] R. K. Cunningham, R. P. Lippmann, D. J. Fried, S. L. Garfinkel, I. Graf, K. R. Kendall, S. E. Webster, D. Wyschogrod, and M. A. Zissman, "Evaluating intrusion detection systems without attacking your friends: The 1998 DARPA intrusion detection evaluation," in *SANS 1999*, 1999.

[20] D. D. Clark and D. L. Tennenhouse, "Architectural considerations for a new generation of protocols." in *SIGCOMM*, 1990, pp. 200–208.

[21] S. McCanne and V. Jacobson, "The BSD packet filter: A new architecture for user-level packet capture," in *Proc. of the 1993 Winter USENIX conference*, Jan. 1993.

[22] A. V. Aho and M. J. Corasick, "Efficient string matching: an aid to bibliographic search," *Communications of the ACM*, vol. 18, no. 6, pp. 333–340, June 1975.

[23] H. Hyyrö and G. Navarro, "Bit-parallel witnesses and their applications to approximate string matching." *Algorithmica*, vol. 41, no. 3, pp. 203–231, 2005.

[24] J.-M. Champarnaud, F. Coulon, and T. Paranthoën, "Compact and fast algorithms for safe regular expression search." *Int. J. Comput. Math.*, vol. 81, no. 4, pp. 383–401, 2004.

[25] L. Tan and T. Sherwood, "A high throughput string matching architecture for intrusion detection and prevention," *isca*, vol. 00, pp. 112–122, 2005.

[26] Z. K. Baker and V. K. Prasanna, "Time and area efficient pattern matching on FPGAs," in *FPGA '04: Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays*. New York, NY, USA: ACM Press, 2004, pp. 223–232.

[27] C. R. Clark and D. E. Schimmel, "Scalable pattern matching for high speed networks," in *FCCM '04: Proceedings of the 12th Annual*

IEEE Symposium on Field-Programmable Custom Computing Machines. Washington, DC, USA: IEEE Computer Society, 2004, pp. 249–257.

[28] S. Kumar, S. Dhamapurikar, F. Yu, P. Crowley, and J. Turner, "Algorithms to accelerate multiple regular expressions matching for deep packet inspection," in *Proc. of Sigcomm 2006*, Pisa, Italy, Sept. 2006.

APPENDIX

The filters we use for our benchmarks were selected to evaluate all aspects of Ruler performance independently. All filters assume a packet at the hardware level. That is, including the Ethernet header.

The filters contain an include command (not shown) that loads a list of layouts for common packet layouts. For example, `Ethernet_IPv4` is a pattern for the layout of an Ethernet header, with the correct protocol number for IPv4 in the protocol field.

The **aa** filter simply accepts all input:

```
filter aa
  * => accept;
```

It provides a baseline for the overhead of the filtering infrastructure. The **ha**, **hi**, and **hr** filters shown below all select a packet based on the header of the packet. In particular, they accept all UDP packets over IPv4 to port `0x7AAA`, and reject other packets. The filters differ in the action they take for accepted packets.

The **ha** filter simply accepts packets.

```
filter ha
  eh:Ethernet_IPv4 ih:IPv4
  UDP with [dest=0x7AAA~2]
  p:*
   => accept ;
```

The **hi** filter zeroes the Ethernet and IP source and destination address, and the UDP destination port. This rewrite operation does not require the entire packet to be copied: it is implemented by only modifying selected elements of the input packet buffer.

```
filter hi
  eh:Ethernet_IPv4 ih:IPv4
  uh:UDP with [dest=0x7AAA~2]
  p:*
    =>
  eh with [e_dest=0#6,e_src=0#6]
  ih with [src=0,dest=0]
  uh with [dest=0]
  p;
```

The **hr** filter reverses the order of the packet headers and payload. This requires all bytes in the packet to be copied to an output buffer (not practical, but interesting as a benchmark):

```
filter hr
  eh:Ethernet_IPv4 ih:IPv4
  uh:UDP with [dest=0x7AAA~2]
  d:*
    =>
  d uh ih eh ;
```

Similar to the previous group of filters, each of the filters **pa**, **pr**, and **pr** select the same packets. In this case they all reject packets with the string `REJECT` in the payload. For packets that don't contain this string, every byte must be inspected, making these filtering operations potentially much more expensive. Again the filters differ in the action they take for accepted packets. The **pa** filter simply accepts the remaining packets:

```
filter pa
  eh:Ethernet_IPv4 ih:IPv4 uh:UDP
  * "REJECT" *
    =>
  reject;
  * => accept;
```

The **pi** and **pr** filter have rewrite actions on the remaining packets similar to the **hi** and **hr** filters respectively.

The **la** filter accepts or rejects packets. Its patterns were deliberately chosen to require a large number of states. This filter has by far the largest number of states, almost 20,000:

```
filter la
  eh:Ethernet_IPv4 ih:IPv4 uh:UDP
  * (0x00 byte#8 "z") *
    =>
  reject;
  eh:Ethernet_IPv4 ih:IPv4 uh:UDP
  * (0xff byte#8 0x11) *
    =>
  reject;
  eh:Ethernet_IPv4 ih:IPv4 uh:UDP
  * ("R" byte#4 "T") *
    =>
  reject;
  * => accept;
```

The remaining filters are representative of real filtering and anonymisation policies. The **all** filter is a typical general-purpose anonymisation filter. It selects all IP and ARP traffic. It zeroes addresses, and removes the payload entirely:

```
filter all
  eh:Ethernet_IPv4 ih:IPv4 d:*
   =>
  eh with [e_dest=0#6,e_src=0#6]
  ih with [src=0~4,dest=0~4];

  eh:Ethernet_ARP ah:ARP d:*
    =>
  eh with [e_dest=0#6,e_src=0#6]
  ah with [sender_ha=0#6,sender_ip=0#4,
           target_ha=0#6,target_ip=0#4];
```

The **www** filter is representative of a more refined selection and anonymisation filter. It selects all TCP traffic to port 80 (the standard HTTP port), and rewrites it to a version with zeroed source and destination addresses, but untouched payload:

```
filter www
  eh:Ethernet_IPv4 ih:IPv4_TCP
  th:TCP with [dest=80~2]
  p:*
     =>
  eh with [e_dest=0#6,e_src=0#6]
  ih with [src=0~4,dest=0~4]
  th p ;
```

The **snort** filter implements the `backdoor` rule set from Snort [7]. The set consists of 77 rules, of which we can translate 66 to equivalent Ruler rules. The remaining rules require information about the TCP flow ('`flowbits`'), which is beyond the scope of Ruler. The filter is too large to show.