Packet Classification Through Regular Expression Matching on NetFPGA

Gianni Antichi
Dept. of Information Engineering
University of Pisa
Pisa, Italy
gianni.antichi@iet.unipi.it

Andrea Di Pietro
Dept. of Information Engineering
University of Pisa
Pisa, Italy
andrea.dipietro@iet.unipi.it

Domenico Ficara
Cisco System International
Rolle, Switzerland
dficara@cisco.com

Stefano Giordano
Dept. of Information Engineering
University of Pisa
Pisa, Italy
s.giordano@iet.unipi.it

Gregorio Procissi
Dept. of Information Engineering
University of Pisa
Pisa, Italy
g.procissi@iet.unipi.it

Fabio Vitucci
Dept. of Information Engineering
University of Pisa
Pisa, Italy
fabio.vitucci@iet.unipi.it

ABSTRACT

The process of classifying packets according to a set of general rules is crucial to many network functions, from QoS enforcement and network monitoring to security and firewalls. Although classification is a well studied subject, most of the literature is concerned with matching prefix based rules over the canonical 5-tuple of packet meta-data. However, we believe that future applications (e.g. layer 7 monitoring) would benefit from an increased flexibility in the definition of classification rules. Indeed, the main contribution of this paper is a novel classification method that applies pattern matching techniques (which are already widely used in the field of deep packet inspection due to their high expressiveness), to traffic classification, by specifying a rule as a pattern over a stream composed by the packet meta-data. We implemented a high performance prototype of our scheme designed for NetFPGA boards. As these devices provide a limited amount of memory, we take advantage of a very compressed version of Deterministic Finite Automata (DFA), that combines excellent compression and high speed.

Keywords

High Performance, Packet Classification, DFA, FPGA

1. INTRODUCTION

The rapid growth of Internet and the fast emergence of new network applications have brought great challenges and complex issues in deploying high-speed and QoS guaranteed IP network. For this reason packet classification has assumed a key role in modern communication networks in order to provide security and QoS. As packet classification represents one of the most important and critical function in the process of IP packet forwarding, over the years there have been many contributions in this area of research. However, classification rules have been generally defined as prefixes over the canonical 5-tuple. Indeed, the industry standard in rule specification is Cisco ACL, which essentially deals with addresses, masks and ports (although a more expressive extension has been proposed). However, modern applications might need additional flexibility in the definition of classification rules. As an example, consider a network security application that detects TCP Syn flooding attacks: such software would be interested in being forwarded by a classifier only the tcp segments carrying a given combination of flags: this could be done by specifying a rule that takes also the TCP flags into account. A standard 5-tuple based classifier, on the other hand, would pass it all of the TCP traffic matching the rules, thus forcing the application to process a much higher amount of packets, most of them of no interest for its purpose. In order to extend the expressiveness of standard classification rules, we chose to address the packet classification problem as a more general pattern matching problem. To this end, we leveraged the existing work on finite state automata (henceforth DFAs). DFAs are systems derived from the theory of formal languages, capable of scanning an orderly string of symbols in search of specific patterns. They are currently widely used for deep packet inspection because they guarantee by far greater expressiveness than traditional techniques. As classification is often performed by either hardware or embedded processors, where memory footprint is an important issue, we have chosen for our scheme a new type of automaton, called δFA (δ Finite Automata), which presents interesting performance characteristics. In particular, in addition to maintaining a data structure which is much more compact than the standard automation, it needs a lower number of memory accesses than most compressed automata. These properties motivate its use for classification as high-capacity networks, where memory latency is the main factor of performance degradation. In order to implement a prototypal classifier, the hardware platform used is NetFPGA. A NetFPGA card is a PCI card consisting of a fully programmable FPGA (Field Programmable Gate Array), four Gigabit Ethernet ports and a set of memory banks with different densities and latencies (internal cache, SRAM, DRAM). This device is employed by students and researchers to experiment new networking features at line speed. This board has been designed by Stanford University as part of a project called Clean Slate, an ambitious research program that aims at producing solutions to overcome the current limitations of the Internet. The FPGA is programmed by means of a hardware description language (HDL) which, in our case, is Verilog.

2. RELATED WORKS

Packet classification is an extensively studied topic and several different approaches have been proposed in the literature.

Hardware classifiers traditionally used CAM based techniques. Given an input key, a Content Addressable Memory (CAM) compares it against all of the memory words in parallel; hence, a lookup effectively requires one clock cycle. While binary CAMs perform well for exact match operations, the widespread use of CIDR requires storing and searching entries with arbitrary prefix lengths. Hence, Ternary CAMs were developed with the ability to store an additional Don't Care state thereby enabling them to retain single clock cycle lookups for arbitrary prefix lengths. This high degree of parallelism comes at the cost of storage density, access time, and power consumption.

A few solutions tried to leverage longest-prefix matching trie-based algorithms (which were conceived for lookup applications) to bi-dimensional matching involving several fields. Such solutions [4] are typically adopted when rules are specified only over destination and source IP addresses. The set-pruning algorithm provides good lookup speed but its memory footprint explodes with the number of rules. A variation of this technique leverages a backtracking primitive [12] to improve memory scalability, at the cost of a significant slow-down, while Grid of Tries [14] fairly balances speed and memory consumption, by speeding up backtracking through the use of switch pointers.

Other solutions leveraged a geometric formalization of the classification problem [1]: as each classification rule can be thought as a range in the multi-dimensional space, classifying a packet means finding out which ranges the corresponding point belongs to. To this end, well-known results from the field of computational geometry can be used.

Another approach is to split a multi-dimensional matching into a set of separate one-dimensional searches over separate fields and to correlate those result to find the best matching rule. Bitvector linear search [13] performs such correlation by processing bitmaps issued by each separate single-field matching. Cross-product schemes [19], instead, correlate the result of mono-dimensional matching by building a cross product hash table where all the combination of mono-dimensional ranges are precomputed. Such a table, however, can grow up to considerable sizes, thus requiring an excessive amount of memory. Recursive flow classification [9] allows to compress such a table by merging the cross-products into equivalent classes.

Another class of algorithms leverage decision trees: although, formally, the algorithm model is analogous to the trie-based approaches, it allows for larger flexibility, as,instead of having all of the relevant fields inspected in a sequential manner, at each node of the tree an arbitrary check can be performed. In particular, Hicuts [8], performs a range check on a particular field while [20] tests single bits. Hypercuts [6] further improbe performance by checking multiple fields at each step. [3] proposed to optimize decision tree by introducing the common branches optimization: rules that, due to wildcards, are assigned to both sons of a decision node, are handled separately, thus reducing worst-case size. [5] Proposes to speed up classification by using a small cache using a set of evolving rules which preserve classification semantics. [17] Partitions the rules into sets which are close to one another in the tuple space, and leverages information from single-field lookups to discard subsets and limit the search space.

3. NETFPGA BOARD

NetFPGA is a low-cost platform, developed by the High Performance Networking Group at Stanford University, primarily designed as a tool for teaching networking hardware and router design. It is a standard PCI card that plugs into a standard PC. The card contains a Field Programmable Gate Array (FPGA) by Xilinx (Virtex-II pro) which is programmed with user-defined logic and has a clock of 125 MHz. The PCI interface connecting the host PC to the NetFPGA is managed by a small Xilinx Spartan II FPGA. Four 1GigE ports, 4.5MB of Static Ram (2 banks) and 64MB of DDR2 Dynamic RAM are also on board in the card. A reference package containing verilog source code for the FPGA, C code for the host PC and java code for the graphical interface can be downloaded from the NetFPGA website in order to run NetFPGA with basic networking functions such as Network Interface Card

(NIC), PW-OSPF IPv4 Router and Layer 2 switch. The basic target for this board is the adoption of a FPGA as a networking accelerator in order to take advantage of the host PC flexibility to implement the control plane of the project. In this scenario, for example, the user could implement the forwarding plane of an IPv4 router in FPGA and the control plane (with its routing algorithm) in the host PC connected to the card via PCI. Thanks to its modularity, NetFPGA is a very useful system to test new ideas for next generation networks.

4. PACKET CLASSIFICATION AND PATTERN MATCHING

The operation of classifying IP packets depending on arbitrary metadata contained in the packets themselves is logically (and practically) equivalent to perform *pattern matching*. Typically, classification rules are expressed in terms of the values of the canonical 5-tuple $SrcIP$, $DestIP$, $SrcPort$, $DestPort$, and $L4-Protocol$: the output of classification can therefore be obtained by simply applying pattern matching algorithms upon the associated fields of the IP packets. However, our scheme supports classification rules defined over arbitrary metadata (TCP flags are a simple example). As pattern matching is a widely addressed topic in literature, the above observation opens a wide horizon of theoretical and practical solutions to address the problem of packet classification. In recent years, due to the increasing interest focused on *deep packet inspection*, the use of regular expressions (regexes) has become more and more popular because of their high expressiveness in describing sets of strings [16]. Typically, *finite automata* are employed to implement regular expression matching. Deterministic FAs (DFAs), in particular, have gained significant credits as they require one state traversal per character only, although they need an excessive amount of memory as the number of regexes increases. For these reasons, many works have been recently presented with the goal of memory reduction for DFAs, by exploiting the intrinsic redundancy in regular expression sets [11, 10, 2, 15].

In this section we briefly present the structure of a DFA along with the principles at the basis of its space compression. In particular, we present δFA [7], a technique inspired by Kumar's work [11] that we recently developed to effectively reduce the memory footprint of large DFAs. As it will be elaborated upon in the following, δFAwill be the basis for the packet classifier implemented on top of the NetFPGA platform.

4.1 From DFA to δFA

We introduce the principles of DFA and δFA [7] by analyzing the same example brought by Kumar et al. in [11]: figure 1(a) represents a standard DFA on the alphabet $\{a, b, c, d\}$ that recognizes the regexes (a^+),(b^+c) and (c^*d^+).

Figure 1(b) shows the D^{2}FA for the same set of regular expressions [11], where the memory footprint of states is reduced by storing only a limited number of transitions for each state and taking a default transition for all input char for which a transition is not defined. The total number of transitions was reduced to 9 (less than half of the equivalent DFA which has 20 edges), thus achieving a remarkable compression.

However, by observing the graph in figure 1(a), it is evident that most transitions for a given input lead to the same state, regardless of the starting state; in particular, adjacent states share the majority of the next-states associated with the same input chars. Then if we jump from state 1 to state 2 and we "remember" (in a local memory) the entire transition set of 1, we will already know all the transitions defined in 2 (because for each character they lead to the same set of states as 1). This means that state 2 can be described with a very small amount of bits.

The result of what we have just described is depicted in fig. 1(c) (except for the local transition set), which is the δFA equivalent to the DFA in fig. 1(a). We have 8 edges in the graph (as opposed to the 20 of a full DFA) and every input char requires *a single state traversal* (unlike D^{2}FA).

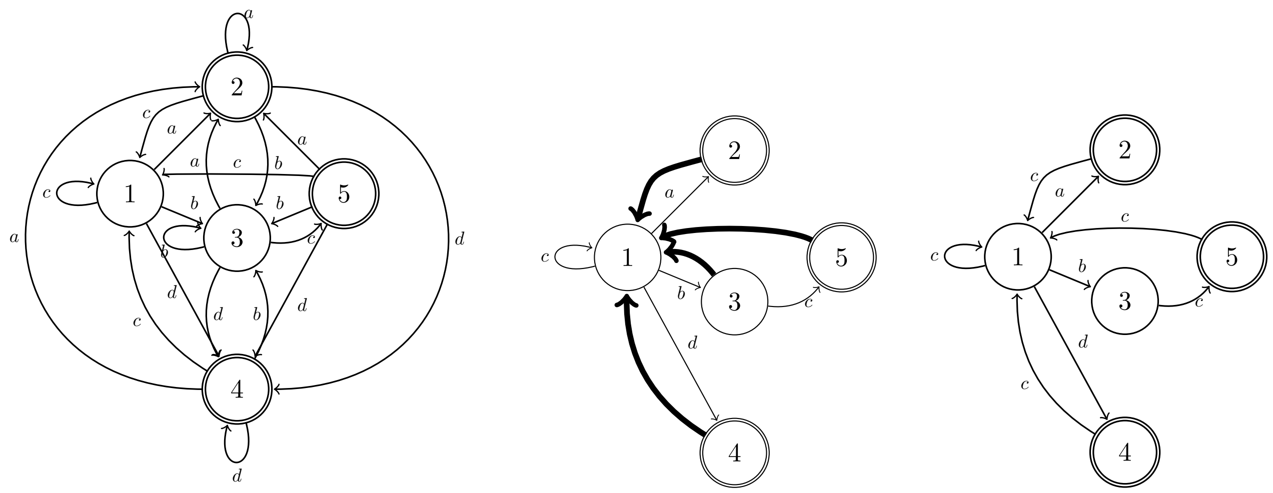
4.2 The main idea of δFA

The idea of δFA comes from the following observations: 1) a state is defined by its transition set and by a small value signalling if it is an accepting state; 2) in a DFA, most transitions for a given input char are directed to the same state.

By elaborating on the last observation, it becomes evident that most adjacent states share a large part of the same transitions. Therefore we can store only the differences between adjacent states.

This requires, however, the addition of a supplementary structure that locally stores the transition set of the current state. The idea is to let this local transition set evolve as a new state is reached: if there is no difference with the previous state for a given character, then the corresponding transition defined in the local memory is taken. Otherwise, the transition stored in the state is chosen. In all cases, as a new state is read, the local transition set is updated with all the stored transitions of the state.

The δFA shown in figure 1(c) only stores the transitions that *must* be defined for each state in the original DFA. In order to improve memory reduction (at the expenses of a negligible increase in the lookup time), δFA can also be efficiently combined with an encoding scheme for transitions (named *Char-State* compression [7]), which exploits the association of many states with a few input characters.



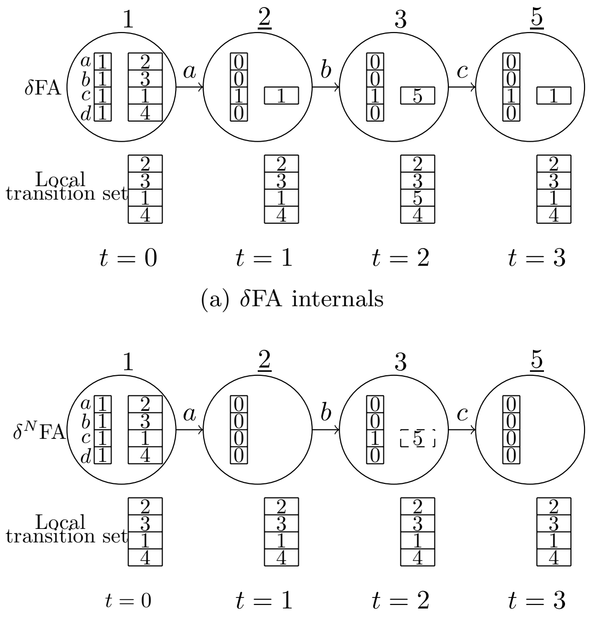
(a) The DFA (b) The D^2FA (c) The δFA

Figure 1: Automata recognizing (a^+),(b^+c) **and** (c^*d^+)

4.3 Lookup

In the first step of the lookup process, the current state must be read with its whole transition set. Then it is used to update the local transition set: for each transition defined in the set read from the state, we update the corresponding entry in the local storage. Finally the next state is computed by simply observing the proper entry in the local storage. The lookup algorithm requires a maximum of C elementary operations (such as shifts and logic AND or popcounts), one for each entry to update. However, in our experiments, the number of updates per state is around 10. Even if the actual processing delay strictly depends on many factors (such as clock speed and instruction set), in most cases, the computational delay is negligible with respect to the memory access latency.

In fig. 2(a) we show the transitions taken by the δFA in fig. 1(c) on the input string *abc*: a circle represents a state and its internals include a bitmap (as in [18] to indicate which transitions are specified) and the transition set. The bitmap and the transition set have been defined during construction. We start ($t = 0$) in state 1 that has a fully-specified transition set. This is copied into the local transition set (below). Then we read the input char a and move ($t = 1$) to state 2, that specifies a single transition toward state 1 on input char c. This is also an accepting state (underlined in figure). Then we read b and move to state 3. Note that the transition to be taken now is not specified within state 2 but it is in our local transition set. Again state 3 has a single transition specified, that this time changes the corresponding one in the local transition set. As we read c we move to state 5 which is again accepting.



(a) δFA internals

(b) δ^NFA internals

Figure 2: Automata internals: a lookup example.

5. δFA BASED PACKET CLASSIFIER

5.1 Software

The software level of the Classifier takes care of creating and managing the DFA data structures as well as of storing them into the NetFPGA SRAM. The user has to write a simple text file to specify the rules and the associated flowIDs. A bash script then is in charge of calling all the software in order set up the Classifier. First of all it creates the standard (uncompressed) DFA associated to the rules specified by the user. After that it converts the DFA into the δFA structure.

In general, a δFA state consists of a bitmap of 256 bits, which indicate which transition are stored, followed by a list of pointers for such transitions. If the number of transitions in the state is small enough, the bitmap is not an optimal solution, because it would be composed almost entirely of bits to "0". In this case it is more efficient (in terms of memory occupancy) to replace the bitmap with a simple flat list of character-pointer pairs. In figures 3 and 4 the data structures of the states of type 1 (we refer to the states with bitmap as type 1 states) and type 2 are shown. Each line corresponds to a 72 bits entry. If the state data do not cover the whole row, the entry is padded with 0s and the new state starts at the beginning of the next line.

S_1 represents the memory occupancy (in bit) of a type 1 state, while S_2 indicates the memory occupancy of a type 2 state where parameter "n" is the number of specified transitions for a given state. As it is evdident, in a δFA the size of a state is not constant because an arbitrary number of transitions may be stored, depending on the characters whose transitions have to be updated in the local table.

$$S_1 = 360 + 72 * \lceil \frac{n}{3} \rceil \tag{1}$$

$$S_2 = 72 + 72 * \lceil \frac{n}{2} \rceil \tag{2}$$

If $n \leq 24$, then $S_1 \geq S_2$ and a simple char-transition list is more efficient than a bitmap. For this reason during the creation of the δFA structure, the number of transitions for each state is estimated. If it is than 24 a type 1 state is created, otherwise a character-pointer list (states of type 2) is used.

The state descriptor is a field whose bits have the following meaning:

- Bit 71: if set to 0 it indicates a state of type 1, otherwise type 2;
- Bit 70: if set to 1 it indicates that the state is accepting;
- Bits 69-64: In the type 2 state, they indicate the number of transitions specified. This information is essential to understand where the state ends. 6 bits are sufficient because for this kind of states there are at most 30 transitions. In type 1 states these bits are set to 0.

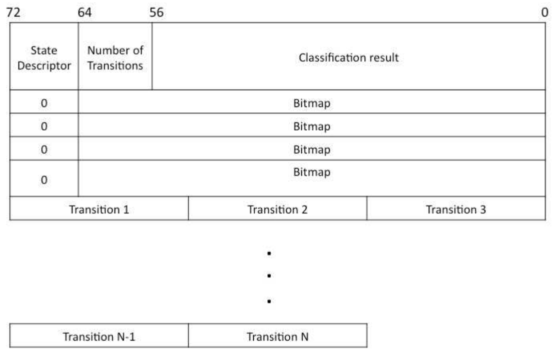


Figure 3: Structure of the classifier.

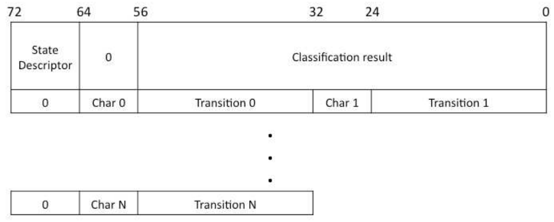


Figure 4: Structure of the classifier.

Type 1 states present also a second byte of information indicating the total number of specified state transitions. This information, wihch is not strictly necessary because it could be derived by counting the total number of bits set in the bitmap, is used to retrieve the data structure through a series of consecutive accesses without having to scan the bitmap. Such bitmap is distributed evenly over 4 rows of 64 bits and an entry contains exactly three pointers.

In type 2 states an entry specifies two transitions, each of them associated with a byte that indicates the corresponding character. The size of type 2 states is from 9 to 144 bytes, while that of type 1 states is from 144 to 819 bytes.

5.2 Hardware

The general structure of the classifier is shown in figure 5. An optimized version will be discussed in 5.2.1 The first operation performed on the incoming packet is parsing the header fields of the packet in order to compose the string which will be fed into the DFA state machine. The "Datapath Control" block extracts the right fields (i.e.: in the implemented prototype the canonical 5-tuple composed of source and destination IP addresses, layer 4 source and destination port and proto-

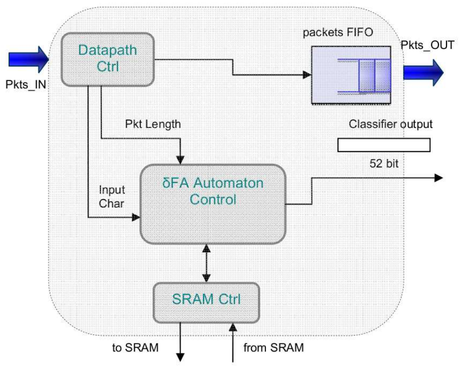


Figure 5: Structure of the classifier.

col) and feeds them, one character per time, into the control module of δFA. The "δFA Control Automaton" block contains the FPGA hardware modules that actually implement the automaton logic (i.e.: extract from the SRAM memory the data structure describing the current state, lookup and update the local transition table). This block communicates with the SRAM via a module that masks the access protocol to the memory and requires as inputs only the address of the first entry to be accessed and the number of consecutive entries to be read. "Sram Ctrl" deals with making the appropriate number of SRAM read/write requests to the "SRAM driver".

The "δFA Control Automaton" module first performs a single access to SRAM in order to determine which kind of states (i.e.: type 1 or 2) it has to read. The local table maintains the current state transitions that are not stored in SRAM, as they are the same as those of the parent state. In its simplest form is a $256 * 24$ matrix where the i-th row contains the transition (a $24 - bit$ pointer) associated with character i. The table is implemented by using Block RAM (BRAM), a type of memory for quick access, integrated on the FPGA chip. In particular, we use a "dual-port" BRAM in "read-first" mode, which allows to write two entries in the table within a single clock cycle. Notice that, for type 2 states, a dual port configuration is enough to avoid buffering transitions, as at most two pointers are extracted simultaneously form the SRAM. As for type 1 states, since three transitions may be read, one of them would have to be buffered and served in the following clock cycle. For this reason, we chose to deploy two dual port "128x24" BRAMs, each of them containing half the original table. The "BRAM 1" stores the addresses for the even characters while the BRAM 2 those associated with the odd characters. In the worst case all of the pointers may still be stored in the same BRAM block and an intermediate buffer would still be necessary, but in the average case the transitions are divided equally between the tables and therefore can be written in parallel during the same clock cycle, thus speeding up the process of updating the local table.

5.2.1 *Optimized Classifier*

An ordinary way to speed up a classifier is caching flows. Packets of the same flow are likely to exhibit good temporal locality, and the classification result issued for the opening packet can be cached and used for the following ones. Therefore it is useful to introduce a flow-cache, where a new entry is added when the first packet of a new flow enters the system. In this case, the classifier performs a lookup in the classifier table and stores the result in the flow cache. Otherwise, for each packet belonging to a known flow, the classification result is already in the cached data and the amount of memory accesses is reduced. Since the number of flows can be very high, a hash table is an efficient way to implement such a cache. In our current implementation, such a table is kept in BRAM memory.

6. EXPERIMENTAL RESULTS

In order to assess the performance of our architecture and its capability to filter traffic at line rate, we carried out several tests by using a Spirent AX 4000 hardware based traffic generator; such a device is able to completely saturate a Gigabit link with minimum sized packets, thus recreating the worst case scenario for a netwrok device performing packet-by-packet processing; actually we always performed our tests with minimum sized packets. As the performance of the classifier is strictly dependent not only on the packet rate, but on the number of flows, which, in turn, reflects on the speed-up introduced by the cache, we used the generator API in order to produce a high number of flows. In particular, the AX 4000 generator can inject packets whose addresses are randomly selected within user defined ranges, thus producing traffic where different flows are randomly interleaved. We point out that this scenario is probably more challenging than real traffic, as in the latter packets from the same flows are close to each othe and a cache can provide a significant speed up. In a first experiment, we used the classifier to extract from the traffic a set of 65536 flows matching properly written regexes. The background traffic (which is simply dropped by the classifier after regex matching) is made up of packets whose addresses are chosen within a very large set (thousands of possible source addresses and as many destination addresses) thus potentially providing milions of different flows. We kept the rate of the traffic of interest constant, while gradually increasing that of the background traffic and we measured the rate of the NetFPGA output by using the capturing facilities provided by the Spirent AX. The results are shown in figure 6 and apparently the classifier manages to filter in all of the traffic of interest with negligible losses. Be-

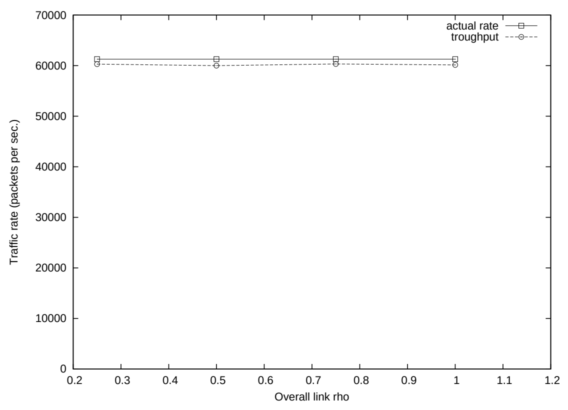


Figure 6: Throughtput of the classifier with constant-rate traffic of interest and different rates of the background traffic (ρ stands for link utilization).

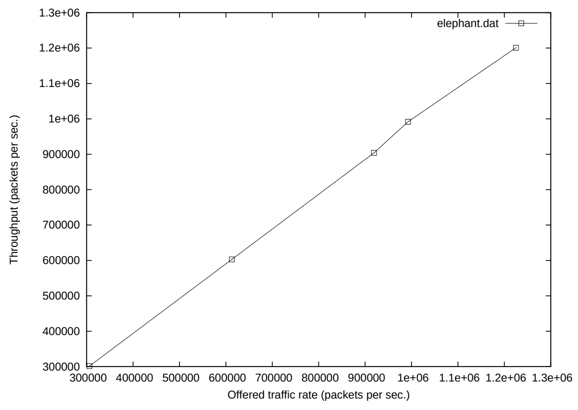


Figure 7: Throughtput of the classifier with growing rates of the traffic of interest.

sides, the performance is almost constant whatever the rate of the background traffic.In a second experiment we assumed all of the incoming traffic to match the classification ruleset, and we increased its rate until the link was completely saturated. Again, as illustrated in figure 7 our classifier is able to process all of the packets with negligible losses.

7. CONCLUSIONS

In this work we described a novel solution for packet classification which leverages the expressiveness of regular expressions in order to specify arbitrary patterns over a set of fields of a packet. Such approach provides additional flexibility with respect to traditional prefix based rule specification. In order to reduce the state required by pattern matching algorithms, we leverage the δFA technique, that provides an excellent trade-off between memory footprint and lookup speed. We built a hardware-based protoypal implementation of our classifier over a NetFPGA board and we assessed its performance by carrying out experiments with high speed traffic belonging to a large set of flows. The results are very promising, as the classifier is able to process all of the packets even in the case of a Gigabit link saturated with minimum-sized packets.

8. ADDITIONAL AUTHORS

9. REFERENCES

[1] H. Adisheshu. Services for next-generation routers. *Ph.D. dissertation, Washington University Computer Science Department*, 1998.

[2] M. Becchi and P. Crowley. A hybrid finite automaton for practical deep packet inspection. In *Proc. of CoNEXT '07*, pages 1–12. ACM, 2007.

[3] E. Cohen and C. Lund. Packet classification in large isps: design and evaluation of decision tree classifiers. In *SIGMETRICS*, pages 73–84, 2005.

[4] D. Decasper, Z. Dittia, G. Parulkar, and B. Plattner. Router plugins: A software architecture for next generation routers. In *IEEE/ACM transactions on Networking*, pages 229–240, 1998.

[5] Q. Dong, S. Banerjee, J. Wang, and D. Agrawal. Wire speed packet classification without tcams: a few more registers (and a bit of logic) are enough. *SIGMETRICS Perform. Eval. Rev.*, 35(1):253–264, 2007.

[6] S. S. F. Baboescu and G. Varghese. Packet classification for core routers: Is there an alternative to cams? *Proceedings IEEE INFOCOM*, 2003.

[7] D. Ficara, S. Giordano, G. Procissi, F. Vitucci, G. Antichi, and A. DiPietro. An improved dfa for fast regular expression matching. *SIGCOMM Comput. Commun. Rev.*, 38(5), 2008.

[8] P. Gupta and N. Mckeown. Design and implementation of a fast crossbar scheduler. *IEEE Micro*, 19:20–28, 1998.

[9] P. Gupta and N. McKeown. Packet classification on multiple fields. *Proceedings ACM SIGCOMM*, 1999.

[10] S. Kumar, B. Chandrasekaran, J. Turner, and G. Varghese. Curing regular expressions matching algorithms from insomnia, amnesia, and acalculia. In *Proc. of ANCS '07*, pages 155–164. ACM.

[11] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner. Algorithms to accelerate multiple regular expressions matching for deep packet inspection. In *Proc. of SIGCOMM '06*, pages 339–350. ACM.

[12] G. V. L. Qiu and S. Suri. Fast firewall implementations for software- and hardware-based

routers. *Proceedings of the 9th International Conference on Network Protocols (ICNP)*, 2001.

[13] G. Malan and F. Jahanian. An extensible probe architecture for network protocol measurement. *Proceedings ACM SIGCOMM*, 1998.

[14] S. S. P. Warkhede and G. Varghese. Fast packet classification for two-dimensional conflict-free filters. *Proceedings IEEE INFOCOM*, 2001.

[15] R. Smith, C. Estan, and S. Jha. Xfas: Fast and compact signature matching. Technical report, University of Wisconsin, Madison, August 2007.

[16] R. Sommer and V. Paxson. Enhancing byte-level network intrusion detection signatures with context. In *Proc. of CCS '03*, pages 262–271. ACM.

[17] H. Song, J. Turner, and S. Dharmapurikar. Packet classification using coarse-grained tuple spaces. In *ANCS '06: Proceedings of the 2006 ACM/IEEE symposium on Architecture for networking and communications systems*, pages 41–50, New York, NY, USA, 2006. ACM.

[18] N. Tuck, T. Sherwood, B. Calder, and G. Varghese. Deterministic memory-efficient string matching algorithms for intrusion detection. In *Proc. of INFOCOM 2004*, pages 333–340.

[19] S. S. V. Srinivasan, G. Varghese and M. Waldvogel. Fast scalable level-four switching. *In Proceedings of SIGCOMM*, 1998.

[20] T. Woo. A modular approach to packet classification: Algorithms and results. *Proceedings IEEE INFOCOM*, 2000.