A Survey on Regular Expression Matching for Deep Packet Inspection: Applications, Algorithms, and Hardware Platforms

Chengcheng Xu, Shuhui Chen, Jinshu Su, *Member, IEEE*, S. M. Yiu, *Member, IEEE*, and Lucas C. K. Hui, *Senior Member, IEEE*

***Abstract*—Deep packet inspection (DPI) is widely used in content-aware network applications such as network intrusion detection systems, traffic billing, load balancing, and government surveillance. Pattern matching is a core and critical step in DPI, which checks the payload of each packet for known signatures (patterns) in order to identify packets with certain characteristics (e.g., malicious packets that carry viruses or worms). Regular expression is the major tool for signature description due to its powerful and flexible expressive ability. However, this flexibility also brings great challenges for efficient implementation in practice. Despite of hundreds to thousands of empirical proposals, wire-speed matching for large scale regular expressions still remains a big challenge. The gap between the matching throughput and the link speed is widening with the ever-increasing network link speed and pattern scale. This survey begins with a full-scale application background of DPI and technical background of regular expression matching in order to provide a global view and essential knowledge for readers. We then analyze the challenges in regular expression matching originated from the state explosion of finite state automaton used for regular expression matching. The nature of state explosion is analyzed in details, and the state-of-the-art solutions are grouped into categories of methods to relieve state expansion and methods to avoid state explosion, suggestions are also provided for building compact and efficient automata in different scenarios. Furthermore, proposals employing parallel platforms, including field-programmable gate array, GPU, general multi-processors, and ternary content addressable memory, to accelerate the matching process are introduced and thoroughly discussed. We also provide guidelines for efficient deployment for each of these platforms.**

***Index Terms*—Regular expression matching, deep packet inspection, pattern matching, content inspection, survey.**

I. INTRODUCTION

MODERN network services increasingly rely on the processing of payload in packets. These services use signatures identified from the payload to perform load balancing, application protocol identification, traffic billing, quality of service control, and network intrusion detection. Deep Packet Inspection (DPI) is a key component in this identification process. Given a set of predefined patterns, DPI matches them with the payload content byte by byte, and returns whether one or more patterns are identified in the content.

In early days, exact strings were used to describe the patterns in DPI. Classical matching algorithms such as Aho–Corasick (AC) [1], Wu–Manber [2] and SBOM [3] were used for fast string matching. However, as signatures were getting more complicated (e.g., with wildcard characters), using exact strings is no longer an effective representation. Regular expression became popular and has been widely used in network applications and devices. For examples, NIDS (network intrusion detection system) of Snort [4] employs regular expression to describe half of its matching rules. Another NIDS, Bro [5], and the Linux application protocol classifier [6] (L7-filter) directly use regular expression to represent all their rules. In industry, network security devices and hardware accelerators on network processors, such as Cisco's security system [7], Cavium matching engines [8], and matching accelerator on IBM PowerEN processor [9], all support regular expression matching. Regular expression has also been used in many other areas such as text editors, programming languages, search engines, and gene sequence matching.

Finite state machine (FSM, also called finite automaton) is a state machine that can recognize the same language expressed by regular expression. They are equivalent in semantics. FSM is useful in regular expression matching. A FSM consists of a set of nodes called states and a set of directed edges with labels connecting the nodes. There is an "initial" state and a set of "accepting" states. Each accepting state represents one or several signatures (patterns). The matching procedure is executed as follows. It starts from the initial state and the first byte of the payload. Each time it reads a byte from the payload, it jumps to the next state(s) according to labels on the edges coming out from the current state. Any time it reaches some accepting states, we say that this payload matches the corresponding signatures. In most DPI applications, automaton-based pattern matching is still the dominating approach for signature matching. Since every single byte in each payload must be processed and each byte involves one to several memory accesses, the matching procedure in DPI is a computation-intensive and time-consuming task and could be

Manuscript received February 2, 2015; revised October 20, 2015 and March 16, 2016; accepted May 7, 2016. Date of publication May 11, 2016; date of current version November 18, 2016. This work was supported by the National Natural Science Foundation of China under Grant 61379148 and Grant 61202488. *(Corresponding author: Jinshu Su.)*

C. Xu, S. Chen, and J. Su are with the School of Computer, National University of Defense Technology, Changsha 410073, China (e-mail: xuchengcheng@nudt.edu.cn; shchen@nudt.edu.cn; sjs@nudt.edu.cn).

S. M. Yiu and L. C. K. Hui are with the Department of Computer Science, University of Hong Kong, Hong Kong (e-mail: smyiu@cs.hku.hk; hui@cs.hku.hk).

Digital Object Identifier 10.1109/COMST.2016.2566669

a major bottleneck of the whole DPI process. Thus, the overall DPI performance depends much on the pattern matching throughput, i.e., the efficiency of FSM matching.

There are two kinds of FSMs, namely nondeterministic finite automata (NFA) and deterministic finite automata (DFA). NFA and DFA are equivalent in expression power (i.e., given an NFA, we can construct an equivalent DFA accepting the same set of strings and vice versa) but different in processing. The most fundamental difference is that, any DFA state has only one transition going to a certain state for each character while a NFA state may have multiple transitions to different states for the same character. Thus, a DFA has only one active state at any time, whereas the NFA may have multiple active states. As a consequence, DFA has a much better $O(1)$ time processing cost per step versus $O(n^2m)$ processing cost for NFA, where m is the number of regular expressions and n is the average length of a regular expression. On the other hand, as expected, the conversion from NFA to DFA may blow up the number of states (state explosion) resulting in a space consumption which can be as large as $O(2^{nm})$. In contrast, the memory consumption is only $O(mn)$ for the corresponding NFA. In conclusion, NFA and DFA have completely opposite features in space consumption and memory bandwidth requirement.

In practice, due to the advancement in networking technologies (e.g., cloud computing), link speed of core routers can exceed 100Gbps, which poses a great challenge for wire-speed pattern matching. Also, the number of malicious signatures with the increasing number of network applications highly extends the number of patterns to be matched simultaneously (e.g., thousands of patterns), which bring a big challenge to regular expression matching in terms of scalability and storage. The conventional memory-intensive DFA cannot meet the scalability requirement and the computation-intensive NFA cannot satisfy the performance demand. Most of current researches are hunting for a tradeoff between storage and performance. The final goal is to perform the matching as fast as DFA while keeping the storage as small as NFA for handling a large set of regular expressions. On the other hand, parallel platforms in commodity network devices like FPGA, GPU, Multi-core processors, TCAM are also employed to accelerate the matching process. Besides, solutions for this problem should meet the requirement of fast update as rules may be altered frequently. To protect against DOS, each packet should be processed in an online manner. No packets should cause processing delay and congestion.

To achieve the above objectives, hundreds of papers have been published. Though there are some reviews focusing on some particular areas, very few surveys have been presented at a systematic level about this topic. In this paper, we aim at presenting a systematic review of regular expression matching for DPI, showing the state-of-the-art researches, identifying knowledge gaps, and providing guidelines for building efficient and compact components for regular expression matching in DPI.

The rest of the paper is organized as follows. Section II presents application background of deep packet inspection, thus providing a global-view for readers, especially general readers, to better understand this field. Section III provides essential technical details for automaton-based regular expression matching, as well as the goals and challenges in practice. Section IV presents automata optimization, including analyzing the reasons for state explosion, DFA based compression algorithms, and scalable FAs to avoid state explosion. Parallel platforms employed for accelerating regular expression matching are presented in Section V, Section VI provides guidelines of building efficient pattern matching components for DPI applications, and Section VII concludes the paper.

II. Deep Packet Inspection Fundamentals

Though our focus is regular expression matching technologies for DPI, it is necessary to give a full-scale background of DPI for better understanding. In this section, we briefly review fundamental aspects of DPI, including what DPI is, the applications of DPI, how DPI works, and the existing surveys on DPI.

A. What is DPI?

Although deep packet inspection (DPI) technology has appeared more than twenty years, it is not so easy to make a workable definition of DPI. DPI contains many Internet technologies such as firewalls, packet capturing or sniffing techniques which have been around for a very long time. Here, we first consider the definition of DPI in Wikipedia [10], then we make a presentation of DPI from a more popular perspective. We also try to clarify some misconceptions of DPI.

Deep Packet Inspection (DPI, also called complete packet inspection and Information extraction or IX) is a form of computer network packet filtering that examines the data part (and possibly also the header) of a packet as it passes an inspection point, searching for protocol non-compliance, viruses, spams, intrusions, or defined criteria to decide whether the packet may pass through or if it needs to be routed to a different destination, or, for the purpose of collecting statistical information.

DPI, as the name implies, involves inspecting the packets passing a specific network point, which is usually a router or switch, analyzing these packets deeply, and making some decisions based on the inspection results. It is called "deep" inspection because the inspection not only includes the packet headers but also covers the packet payloads. However, a common misconception is that DPI only concerns about the payload parts. A better way to understand DPI is to separate the fundamental functions of DPI from various applications based on it [11], as many novices may confuse DPI with upper level applications, such as bandwidth management, network security, user profiling, etc, which will be discussed in the next subsection. The basic procedures of DPI are *recognition* and *action* which is based on the *recognition* result. Recognition is to find the characteristics hidden in these packets, the characteristics may be application protocols, viruses, worms, or special format data (phone numbers, credit card numbers). Then, actions may be triggered by the recognition results, either logging in network analysis or rigorous

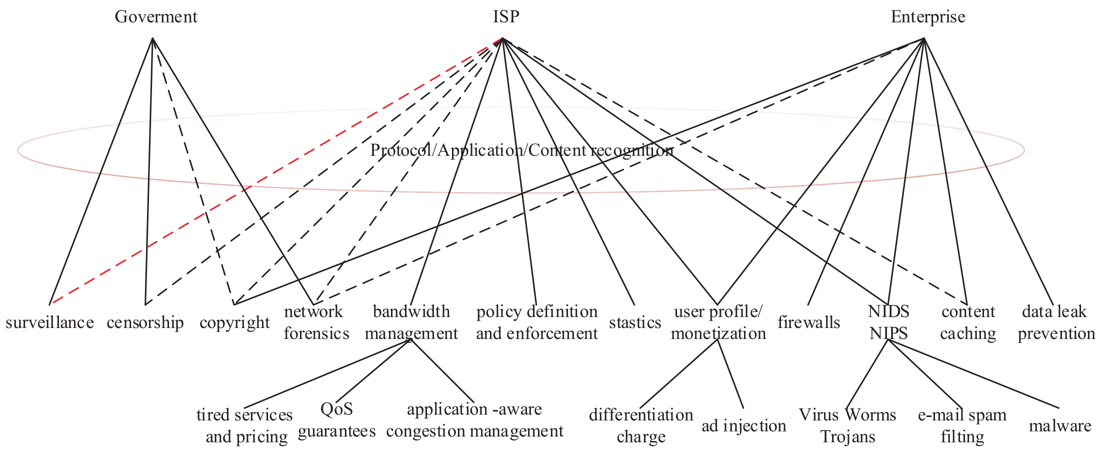


Fig. 1. Various DPI applications from different perspectives, solid line represents primary promoting party, dotted line means secondary promoting party, and red line means illegal activity.

discarding in security applications. Recognition is the foundation, and action is relied on recognition results and associated with specific applications. Both of them will be introduced in the following subsections. The header-payload confusion [12] is another common misconception that is worth mentioning. Packet header contains essential information, such as source and destination addresses, for it to arrive the destination through the network, while the payload carries real data, for instance, files, images, videos, etc. In network architecture, both OSI model and TCP/IP model have multilayered headers, then where exactly the payload starts from the point of view of DPI is not clear. In fact, there is no clear demarcation between header and payload, the data in each level is the header of data in its upper levels. For example, the Ethernet part is the header of IP packet, and IP part is the header of TCP packet. In addition, the inspection starting-point is also related to specific applications.

B. Applications of DPI

Providing an overall classification of DPI applications is a difficult and tedious task, as multifarious recognition methods can be combined with multifarious actions for various applications. Besides, applications in different levels may be mixed up, which will also aggravate the difficulty of classification. Some researches [10]–[14] have tried to classify DPI applications in different levels or from different perspectives. Mueller [11] gave six family groupings of DPI use cases, namely network visibility and bandwidth management, user profiling, governmental surveillance, network security, copyright policing, censorship or content regulation. Bendrath and Mueller [13], [14] summarized similar applications, in addition he listed ad injection as a common use case. Documents in Wikipedia [10] introduce applications from the perspectives of enterprises, internet service providers, and governments. Here, we present classical DPI applications combined with corresponding organizations in Figure 1, namely who deploys DPI for what purposes (applications).

As shown in Figure 1, all DPI applications are based on recognition/identification, say for protocols, L7 applications or content-based signatures. Mueller [11] has summarized the following characteristics to be identified: protocols, applications, URLs, media content, text strings, special format data (e.g., credit card number, phone number), viruses, malware and other cyber-intrusions. Different operators have different requirements and concern different aspects. For examples, ISPs are most concerned about efficiency or money, so bandwidth management is their major interest, enterprises and institutions care more about intranet data security, while governments mainly employ DPI for state security and laws enforcement. Besides, some special applications may involve multilateral participation, for instance, government surveillance and censorship need assistance from ISPs. Next, we will briefly introduce several classical DPI applications associated with their economic, political, or social impacts.

1) Network Security: The original use case for which DPI was developed for is intrusion detection [15]. With the rapid growth of online banking, online shopping and eGovernment, attackers are attracted to break into remote computers or network systems to steal sensitive data for profits. Attackers can leverage weaknesses of the systems to devise sophisticated malware for hacking, such as viruses, worms, spyware, trojans and other malicious codes. As shown in some recent reports [16], [17], malware has increased 26% in 2014. There were more than 317 million new pieces of malware created last year, nearly one million per day. Symantec has observed that malware continues to grow in quality, as well as quantity, and the global infection rate was nearly 36.51% [17]. DPI equipped network operators have the capability to detect known malware before they reach their employees or customers. DPI for network security combines early IDS and latter IPS (intrusion prevention system). IDS detects threats and triggers alerts. IPS takes measures to prevent possible threats (e.g., by terminating the suspected connections). Active IPS is more likely to be deployed by smaller-scale enterprise or institution networks rather than by carrier-grade Internet service providers [11]. Besides outside attackers, hacking the intranet from insiders is also a serious problem. More attention has focused on preventing sensitive data from leaving the intranet, which also called data loss prevention (DLP) [18]. DLP employs DPI technology

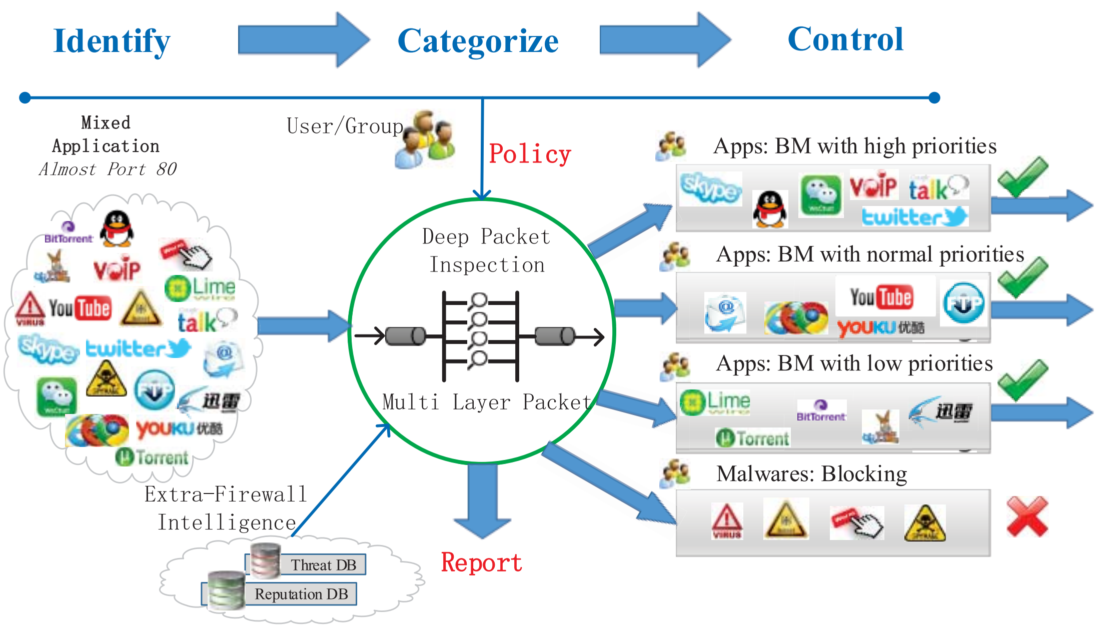


Fig. 2. Procedures and typical applications of DPI, including bandwidth management and network security.

to explore sensitive information hidden in normal streams, and block them from leaving the organizational boundary via some communication protocols. Network forensics [19] is a sub-branch of digital forensics related to capturing and storing information of crimes and instructions for legal evidence or other forms of law enforcement action. Thus, the boundary between network forensics and lawful government surveillance are blur as well.

2) *Bandwidth Management:* As the rapid growth of networking media, file-sharing traffic consumes ever more bandwidth. ISPs have a strong motivation to deploy DPI for bandwidth management [20]. Bandwidth management is the process of controlling the traffic on a network link to avoid poor network performance or network congestion. Bandwidth management relies on traffic classification to classify the traffic into different categories based on the recognized protocols or applications, then applies mechanisms such as traffic shaping, scheduling algorithms, congestion avoidance to each class of traffic differently. Typical uses of DPI-based bandwidth management include prioritizing real-time interactive applications such as online phone calls and online game, rate-limiting bandwidth-intensive applications such as peer-to-peer (P2P) networks, and blocking access to undesired applications such as P2P in an enterprise environment [12]. Figure 2 presents typical procedures and usages of DPI, including network security and bandwidth management. According to the Cisco Visual Networking Index [22], P2P traffic has grown more than 7 petabytes per month in 2014. One of the major missions of DPI based bandwidth management focuses on managing P2P traffic. In particular, P2P file sharing can consume heavy network resources, which will put latency-sensitive services such as VoIP in danger, and this problem will become more serious with more and more HD video streams being transmitted in the networks. On the other hand, P2P also presents potential business opportunity in content distribution network (CDNs) to transfer large files like live streaming media and on-demand streaming media [23]. With DPI's ability of recognizing P2P applications, service providers can leverage DPI to endow latency-sensitive traffics like VoIP or IPTV higher priorities when the network experiences a heavy load, and allow the bursting of P2P applications when the bandwidth is affluent, thus to ensure best QoE for all users. Quality of Service (QoS) guarantee is another type of DPI-based bandwidth management, where ISPs appealing customers to subscribe their services by assuring them that specific levels of quality will be guaranteed for specific types of traffic like VoIP and IPTV. A common allegation against DPI based bandwidth management is that it may breach net neutrality. However, Mochalski and Schulze [12] argued that traffic management brings neutrality to the network as the network should be accessible to every user in a fair manner.

3) *User Profiling/Ad Injection:* With the advent of online marketing and advertising, a more sophisticated and economic method is to push the ads to those who may be interested in them, namely ad injection. Ad injection relies on the knowledge of individual user profile, which can be obtained by tracking and analyzing which websites an user is surfing and what he did in the Internet. For instance, if an advertising company knows that a particular user is interested in electronic products, it can push him ads of mobile phones, laptop computers, Bluetooth headsets even when he is just browsing sport news. Companies like Google leverage cookies, scripts or plugins embedded in browsers to aggregate information for user profiling, then assist advertising companies like Sears and Walmart by inserting relevant ads into users' web page based on their profiles. Yontoo browser plugin earned \$8 million by modifying 4.5 million users' private Facebook sessions for ad injection in 2013 [24]. Thomas *et al.* [25] also revealed that every day about 5.5% of unique IP addresses that visit Google services have at least one installed with ad injector,

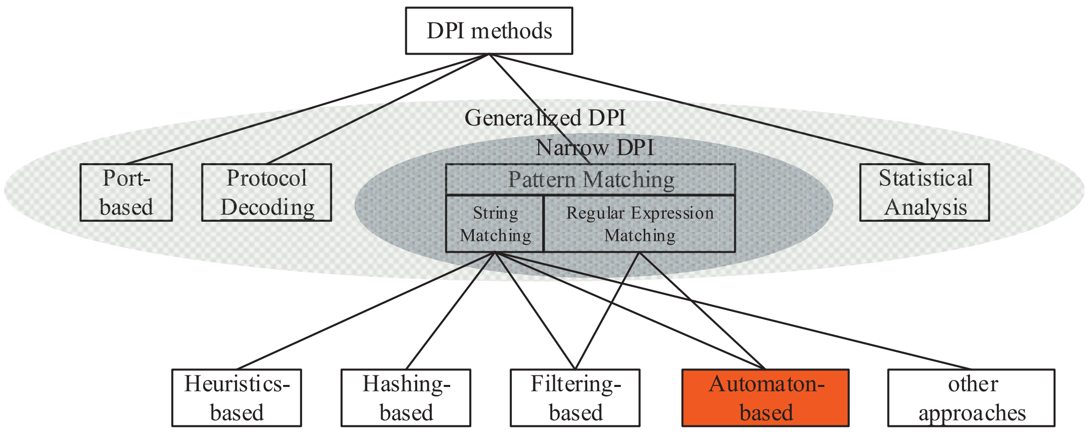


Fig. 3. Popular methods of generalized and narrow DPI, our focus is automaton-based regular expression matching.

and about 50870 Chrome extensions contain ad injectors. On the other hand, as ISPs have the ability to inspect every packet passing through their devices, they also have a strong economic intention to generate comprehensive user profiles with DPI. However, ad injection may affect the customers' browsing experience, privacy and security. Chrome received more than 100,000 complaints in July, 2014, and nearly 20% were about ad injection [25].

4) Copyright Enforcement: The flourishing development of online file sharing has urged the copyright holders such as Sony BMG and Universal Music to seek support from ISPs for copyright enforcement. DPI serves as an online detection tool to detect whether the inspected traffic contains copyrighted materials. Unlike other applications, DPI for copyright inspection cannot employ traditional bit-match or hash methods, as it should be able to recognize the fragments of copyright contents in different media formats or compression levels [26]. Audible Magic has been promoted by the music industry for solving the problem with its fingerprinting technology. Rights holders employ this software to generate a unique signature for each protected material, and store them in a registry. When the media traffic passes through the network, the DPI application calculates a fingerprint for it, and matches this fingerprint with the fingerprinting database registered by rights holders. By the time we are working on this paper, Audible Magic has claimed to have more than 11 million fingerprints of musics, videos and software in its Global Rights Registry [27]. However, ISPs have no motivation or obligation to deploy DPI for copyright enforcement as the returns, if any, only benefit the copyright holders, but seldom for the ISPs. The entertainment industry has tried to solve this issue through lawsuits, interested readers may refer to [13] and [14] for more details.

5) Government Surveillance and Censorship: Almost all governments in the world request communication providers including ISPs to provide surveillance or interception capabilities for law enforcement units and national security agencies. DPI can make a full-scale surveillance of internet traffic as it can catch and analyze anything flowing through a network, and make it known to the governments. For example, the system can record all the activities of a specific IP address, e-mail address, telephone number or other user accounts. However, the government has gone far beyond lawful interception even in U.S., the National Security Agency (NSA) deployed PRISM [28] worldwide to collect activities of any person they are interested in. Bendrath [14] summarized the legal issues and conflicts between the surveillance from U.S. government and the privacy of citizens. Government content censorship is another DPI-based application that blocks illegal contents flowing in the internet. Rather than a specific application or communication protocol, the major concern of internet censorship is the transmitted content. After classification, illegal contents such as pornography and political dissent may be blocked. Due to the difficulty and high cost of real content-based censorship, URL blocking has been the most common from of censorship [11].

C. Methods and Deployment of DPI

In this subsection, we briefly present general procedures and methods of DPI. Then, we will point out our key concern. The generalized concept of DPI covers inspection of both packet header and payload content, where the headers can be used for protocol or application identification with their fixed formats or statistics characteristics, and the payload can be used separately or combined with packet header for content-based recognition, either for traffic classification, NIDS or other purposes. While in a narrow sense, DPI only indicates definite content-based recognition, and the recognition process is implemented by matching the payload with predefined signatures. Referring to the analysis of [29]–[31], we summarize popular DPI methods as in Figure 3. Except the narrow pattern matching method, other generalized DPI methods, namely port-based identification, statistical analysis and protocol decoding are mainly used for traffic classification.

Port-based approach is the most traditional method for protocol identification, which achieves the identification by simply checking the port fields in TCP or UDP headers as IANA [32] assigns well-known port numbers for popular protocols. Due to the high efficiency and immune to encryption, it is widely used for access control lists (ACL) and firewall rules [33]. However, this method is considered to be inaccurate as applications like P2P use random ports, some applications even use ports assigned to other protocols for deception purposes. Reports [34], [35] revealed that port-based methods can only recognize 30%-70% of traffic generated by certain protocols. Currently, it is still employed for identifying applications which always use preassigned ports [36].

Statistical analysis is another payload-independent method for traffic classification. This approach gathers information such as port numbers, packet length, transport layer protocol, inter-arrival time of packets in a flow, flow start and stop timestamp, etc to characterize the traffic and estimate which application or protocol the traffic may belong to. Researchers may collect different aspects of information and devise customised statistical methods [37]–[39] for classification. Machine learning method goes one step further, which combines statistical-based method and heuristics-based algorithms for automatic modeling and analysis. Nguyen and Armitage [40] reviewed related works prior to 2008, interested reader may refer to Finsterbusch's survey [29] for recent machine learning approaches. Roughly speaking, statistical analysis has a higher classification accuracy compared to port-based approach. Statistical method is also insensitive to payload encryption.

Protocol decoding is categorized as a class of traffic classification methods by Finsterbusch *et al.* [29], however, in our opinion it can be viewed as a lightweight pattern matching methodology as it also involves more or less payload inspection. It recognizes protocols by the characteristic protocol headers (magic number, session identifiers, etc) as well as the protocol behavior, which is usually modeled as a protocol state machine. Thus, this method relies on the re-establishment of sessions at application-layer with the captured packets. Though protocol decoding can achieve high accuracy with low false negative rate [41], it requires a deep understanding of the protocol and it is expensive for deployment.

Aforementioned methods are dedicated to traffic classification, on the other hand, methods based on pattern matching cover much more than only traffic classification. Network Intrusion Detection System (NIDS) is a typical application of pattern matching based DPI, which compares the payload content with a set of predefined signatures or patterns to detect malwares like viruses, worms, spyware, trojans hidden in the content. As the positions where signatures may appear are unknown beforehand, it is necessary to implement a thorough matching of each payload byte. Thus it is much more inefficient compared with traffic identification applications, which only need to inspect part of the packet payload or a few packets of a flow. In pattern matching, intuitively, the patterns can be described in exact string format. However, due to the insufficiency in expressive power and flexibility of exact strings, regular expression has become the main representation tool in many applications. Our main focus is also on regular expression matching for DPI. We are now ready to briefly introduce several kinds of algorithms for pattern matching.

Heuristic-based matching algorithms are dedicated for exact string matching, the main idea is to skip as much payload characters as possible according to some heuristics to accelerate the matching process. During the matching, a window with length of m covers the characters waiting to be inspected, and the matching result indicates the next position the window should slide to. The heuristic is to find the appearance of patterns' prefixes in the window's suffix, and determine the next position. Readers may refer to [2] and [42]–[47] for more descriptions and implementation details about heuristic-based matching.

Hashing-based algorithms [48], [49] also do not compare the payload content character by character against the patterns. For each string pattern of length m, a hash value is precomputed. By computing also a hash value for the currently inspected substring of length m in the payload, if it matches the hash value of any pattern (the matching can be done by binary search), the substrings will be extracted and compared with the corresponding pattern byte by byte for verification.

A hashing-based method can quickly judge whether a substring in the payload may match a set of patterns, on the contrary, a filtering-based approach can quickly exclude characters that will definitely not match a pattern. For example, if a segment does not contain any two-character substrings of a pattern, it will never contain this pattern. Snort [4] also employs filtering mechanism for regular expression matching to exclude streams that contain no segment features of a regular expression.

Automaton-based approaches are the most widely used methods for pattern matching, either for string matching or regular expression matching. A finite state automaton is generated for the patterns before matching, the automaton contains an initial state and some final states indicating the matching of some patterns, and other intermediate states represent partial matching situations. The matching process starts from the initial state, each time a payload character is sent to the automaton for state migration. If a final state is visited during the process, the corresponding pattern(s) match is found. The classical automaton-based Aho–Corsick (AC) algorithm [1] was proposed in 1975 for multi-string matching. Although the theoretical time complexity of AC algorithm is independent of the pattern set size, the matching speed will slow down for larger pattern set as the cache locality will become worse for larger state transition table in practice [30]. Some improved AC algorithms [50]–[52] have been proposed to reduce memory requirement for automaton storage. Regular expression matching also relies on finite state automaton, but the powerful and flexible expressive ability brings much more challenge, which we will discuss exhaustively in this paper.

Up to now, we have introduced multifarious applications and methods for DPI, and claimed that regular expression matching is the main focus of this survey. Before we discuss regular expression matching, we provide more information about DPI first. As we have clarified, DPI is a method not an application, different applications implement different deployments and procedures for DPI, also have different abilities and requirements for DPI.

For DPI, the first step is packet capture. There are several hardware combinations like cable splitter and port mirroring for packet capture [53]. Packet capture at hundreds of Mbps is easy with current commodity platforms and softwares, but it is difficult for higher speed like a few Gbps [54]. Antonello *et al.* [54] ascribed this to high CPU load, as the

OS kernel must perform heavy preprocessing tasks for packets before sending them to the user space. Further, the author made a comparison for CPU load among popular APIs, including Libpcap [55], Libpcap-mmap [56], PF_RFING [57] and libe1000 [58]. All these software methods employ bypassing default packet handling mechanism for saving CPU resources when capturing packets from NIC to the user space. Recent proposals such as PFO [59], netmap [60], PF-RING-DNA [61] have shown significant improvement in performance through customizing packet technologies [62]. Intel also released a set of libraries and drivers called data plane development kit (DPDK) for fast packet processing [63]. Schneider *et al.* [64] analysed packet capture challenges in Ethernet environments from hardware aspects, and argued that the inherent memory and system bus throughput is the main limitation. Based on this, he proposed to distribute the traffics of higher speed interfaces to a bunch of lower speed interfaces. In addition, by optimizing the packet processing architecture and providing flexible software development, multicore processors and network processors can also achieve high performance. As our main focus is not packet capture, we refer readers to [65]–[67] for more comprehensive understanding.

Flow-based inspection or packet-based inspection is another issue in DPI deployment. A flow means a set of packets that have the same five-tuple (source IP, destination IP, source port, destination port, protocol number) which can determine a unique session. For TCP protocol, if the data length at the transmitter is bigger than maximum segment size (MSS), then the data should be cut into multiple segments at transport layer before transmission. These segments will be sent separately and reconstructed by the receiver at transport layer. Similarly, for UDP protocols if the data length is bigger than maximum transmission unit (MTU), the data will be cut into multiple packets at network layer during transmission, and these packets will also be reconstructed by the receiver at network layer. This means that logical integrated content is distributed in multiple packets, then the content signatures such as protocols or malware features may locate across multiple packets. Under this situation, only individual packet-based inspection is not able to detect cross-packet signatures. In addition, IP fragmentation is also employed for DOS attacks such as ping of death [68] and teardrop [69] which can cause the system to go down or reboot when performing IP fragments reconstruction. Some routers or NIDSs maybe unable to detect these features or attacks as the lack of IP fragments reorganization ability.

Flow-based inspection is a technique which requires the reconstruction of the packets before they finally reach the receiver, thus flow-based inspection can achieve a higher accuracy than single packet-based inspection. However, flow-based inspection requires much more hardware resources like CPU, memory and bandwidth for DPI system as it needs to maintain the connections among different packets from the same flow. For TCP packets, this is also called TCP reassembly which fabricates the fragmented or disordered packets into an integrated block before sending to the matching engine. Experiments from [70] and [71] both demonstrated that TCP reassembly takes a major part of the whole workload. Thus, there is a tradeoff between inspection accuracy and performance for flow-based inspection. Libnids [72] and Tcpflow [73] are two open source programs performing TCP reassembly. Chen *et al.* [71] also devised a multicore NPU-based TCP stream reassembly card to improve the stream reassembly throughput performance. In a carrier-grade network, the ISP backbone network is a multi-link load-balanced network, and packets of flows are randomly distributed over different links. DPI modules deployed over different links even need extra synchronization to guarantee the accuracy for distributed flows [62]. Though TCP reassembly is related to flow-based deep packet inspection, we do not mean to involve it in this survey as TCP reassembly and pattern matching are separate procedures, and too much discussion may shift the readers' attention. On the other hand, as not all applications require flow-based inspection for high accuracy and our pattern matching dose not aim at a specific application, we will ignore the reassembly process in this survey, but we recommend interested readers to refer to [74]–[76] for more details about TCP reassembly.

In addition, most DPI applications require online real-time recognition of the traffics and make decisions on how to handle them. For instance, the NIDS is required to recognize if a packet or flow contains viruses or worms in real time, and determine whether to forward it or just drop it and generate reports. However, online DPI is much more challenging compared with offline DPI, which captures packets to disks and analyzes them with much longer time constraints. In this survey, we focus on online deep packet inspection.

D. Existing Surveys

In this subsection, we summarize the surveys [29]–[31], [40], [53], [54], [77], [78] on deep packet inspection in recent years, and state clearly the motivations of our survey.

The first three surveys [29], [40], [53] mainly focus on deep packet inspection technologies for Internet traffic classification, not mainly on pattern matching technologies. Nguyen and Armitage's survey [40] focuses solely on statistical analysis for traffic classification, which belongs to the generalized payload-independent DPI method, thus is on a different topic compared to our survey. Callado *et al.*'s survey [53] explains the main techniques and problems in IP traffic identification, it separates the classification methods into packet-based and flow-based categories. Also, this survey also focuses more on the techniques available for traffic analysis, including sampling, signature-matching and inference, and only one paragraph is for signature-matching. Finsterbusch's survey for traffic classification focuses on payload-based approaches, namely the deep packet inspection. The authors present a complete analysis of the most popular open-source traffic classification modules, including OpenDPI, nDPI, IPP2P, HiPPIE, libprotoident and L7-filter. These modules involve pattern matching and protocol decoding approaches for traffic classification, and the evaluation comprises classification accuracy, memory usage, CPU utilisation, etc, thus can provide general guidelines to design and implement these modules.

The following two surveys by Lin *et al.* [30], [31], review pattern matching approaches for deep packet inspection, but they mainly focus on exact string matching not the regular expression matching. Though they share the same automaton-based method, regular expression is far more complex and challenging than exact string matching. Lin *et al.* [30] classified string matching algorithms into four categories according the data structures used in these algorithms. Furthermore, the author analyzed algorithms deployed in three network content security packages (ClamAV, DansGuardian and Snort), and designed an algorithm named CRKBT to enhance the matching performance. Lin *et al.* [31] focused more on the implementation issues of string matching, including the ability to support multigigabit connections and large volumes of signatures, which is also the main challenges in regular expression matching.

There are three surveys [54], [77], [78] that are more similar to our survey. But the first two works [77], [78] are much more lightweighted compared with [54] and our paper. Also, there are many differences between these papers and our survey in terms of references, organizations, categories, and technical perspectives, which we will discuss later in detail. AbuHmed *et al.* [77] reviewed the implementation techniques and challenges of deep packet inspection for NIDS from both software and hardware aspects. Besides, the author also provided a simple comparison among existing implementations till 2007. However, the structure is a little bit confusing and some important materials are missing both in hardware part and software part. And it only surveyed researches before 2007. On the other hand, Rathod *et al.*'s survey [78] only focuses on finite automata algorithms for pattern matching. Although this paper was published in 2014, it only cites 10 papers in total and some references may not represent the development in this area. The author casually divided finite automatons into three categories, which may not be comprehensive and detail enough.

Among all the mentioned surveys, only Antonello *et al.*'s survey [54] published in 2012 has the same focus as our survey and is more comprehensive. In their paper, the author first provided essential technical background, including string matching, regular expressions, finite automaton and the automaton based matching process. Finite automaton is the core technique for regular expression matching, thus leading to a quick understanding of regular expression matching for novices. Subsequently, the author introduced packet capture support from operating system and made a comparison of CPU load among current available APIs for packet capture. Hardware platforms for packet inspection are also mentioned in this part, but it is not detailed enough and not a major emphasis of their paper. However, the advancement on parallel hardware platforms make hardware platform a critical factor in regular expression matching for DPI. This is also a major motivation for our paper. In addition, an important platform called ternary content addressable memory (TCAM) is also missing in their paper. The section named *Optimizing DPI engines* accounts for the core part of their paper, where the authors discussed FA compression algorithms and classified them into three categories, namely FA grouping algorithms, FA transitions compression and FA state space compression. An evaluation of these FA compression techniques is provided in another paper of the same author [79]. Finally, the author concluded the challenges for DPI systems, the challenges are attributed to the increasing signature sets and data exchanging bottlenecks of hardware and operating system. Guidelines are also put forward from hardware level to the user space level in performing a DPI based traffic classification system.

Although we have the same focus, our survey differs from Antonello's survey in a number of ways. First, considering the multifarious applications of DPI, many novices may mix up the applications and DPI itself. We present an exhaustive background to promote understanding, namely which organization deploys DPI at which network level for what purposes. Moreover, we conclude methods used for DPI, and clearly defines our study scope as automaton based regular expression matching. Though these materials have little relation with technical details, they are necessary to equip the novices with a global view. Second, although hardware platform plays an important role in the matching process, none of the existing surveys have discussed it in details. This motivates a comprehensive survey for this topic. In addition, we analyze FA related techniques from a different perspective. We attribute the regular expression matching challenge to state explosion, and analyze the causes of explosion from both RE characteristics and mathematic perspective. Based on whether degrading or eliminating the explosion, we roughly classify the FA techniques into two categories, FA compression methods and scalable FA methods. In each category, we divide them into more fine-grained categories according to our taxonomic method. This is also a main contribution of our survey. Finally, as the latest meaningful survey is published years ago, many important researches have been published in recent years, it is necessary to analyze these new researches to understand the state-of-the-art development in this area. We believe that combining [54] with our survey, readers may achieve a deep and comprehensive understanding on regular expression matching techniques for deep packet inspection.

III. Automaton Based Regular Expression Matching: Technical Background

In this section, we introduce the technical background for automaton based regular expression matching. We first present the notion of regular expression and its use in popular DPI systems. Then, we explain the relationship between regular expression and finite state automaton (FSA or FA, also called finite state machine, FSM), which is the core component of regular expression matching, and illustrate the automaton based matching process. Finally, two traditional FSM named nondeterministic finite automata (NFA) and deterministic finite automata (DFA) are discussed in detail. Additionally, some other matching issues are also presented here.

A. Regular Expression

In theoretical computer science and formal language theory, a regular expression is a sequence of characters that define a search pattern, namely languages or a set of strings. In the normal case, a regular expression over alphabet Σ can represent a set of strings in Σ^* without enumerating them. It

TABLE I
COMMON METACHARACTERS AND CORRESPONDING MEANINGS

Type	Metacharacter	Description	Example
Character class	.	matches any single character	
	[]	matches a single character contained in the brackets	$[abc]$ matches a or b or c
	[^]	matches a single character not contained in the brackets	$[\hat{}ab]$ matches any character other than a or b
Quantification	$*$	matches the preceding element zero or more times	$a*b$ matches b, ab, aab, etc
	?	matches the preceding element zero or one time	$ab?c$ matches ac and abc
	$+$	matches the preceding element one or more times	$a+b$ matches ab, aab, $aaab$, etc
	$\{m,n\}$	matches the preceding element from m to n times	$a\{1,3\}b$ matches ab, aab and $aaab$
	$\{m,\}$	matches the preceding element at least m times	$a\{1,\}b$ matches ab, aab, $aaab$, etc
	$\{m,n\}$	matches the preceding element at most n times	$a\{,3\}b$ only matches b, ab, aab and $aaab$
Position	^	matches the starting position within the string or line	$\hat{}hat$ matches hat but only at the start of the string or line
	\$	matches the ending position of the string or line	$cat\$$ matches cat but only at the end of the string or line
Others	|	OR relationship	$abc\|def$ matches abc or def

was first proposed by Kleene [80] in 1950s and has been widely used in programming languages, text editors, network security, etc.

Each symbol in a regular expression is either a regular ASCII character or a metacharacter representing some special meanings. Unlike an exact string, a regular expression can represent a set of exact strings with the expressive power of metacharacters, while an exact string can only represent one string, i.e., itself. For example, the expression "$.*a+b$" represents any string containing arbitrary number of *a* followed by a *b*. Some of the common metacharacters are listed in Table I. We simply classify the common metacharacters into three categories, namely character class, quantification and position. '.', '[]', and '[^]' belong to the first category, each of them can match a set of characters. For example, a single '.' can match any of the 256 ASCII character. Some other metacharacters not listed in this table can also represent a set of characters, for instance, '\d' matches a digit, '\w' matches an alphanumeric character, '\s' matches a whitespace character, etc. The quantification metacharacters represent to match the preceding element a specified times, '$*$' means 0 to more times, '?' means 0 to 1 time, '$+$' means 1 to more times, etc, as list in the table. The preceding element maybe a character, a string or even a sub regular expression. The other type of metacharacters is position metacharacter, which can specify the matching position. '^' matches the starting position and '\$' matches the ending position of a string or line. The combination of these metacharacters can describe complex meanings. For example, the regular pattern of QQ protocol in L7 is "^.?.?\x02.+\x03\$", which matches the payload starts with 0 to 2 arbitrary characters, followed by the ASCII value 02, then 1 or more arbitrary characters, and ends with the ASCII value 03.

In network applications, regular expression matching is mainly used for application protocol identification and NIDS, and a regular expression pattern may represent the unique characteristic of an application-level protocol, a virus, a spam or a malware. Due to the powerful and flexible description ability of regular expression, it has been widely used in several open source DPI applications and commercial DPI engines, and the trend is still growing. For example, the open source NIDS of Snort [4] involved no regular expression patten in April 2003, to 1131 REs (February 2006), 13605 REs (February 2014), and the proportion is still in growing. Another NIDS, Bro [5] and Linux application protocol classifier (L7-filter) [6] even describe all their patterns with regular expression. In the commercial DPI systems or components, like Cavium matching engines [8], Cisco's security system [7], IBM PowerEN processor [9], etc, all support regular expression matching.

B. Finite State Automaton

Before the introduction of finite state automaton, we provide some basic concepts about alphabet, string, language and regular language. The *alphabet* is a nonempty, finite set of symbols, here it refers in particular to the 256 ASCII code, and we often use symbol Σ for representation. A *string* is a finite sequence of symbols chosen from a alphabet Σ and a language is a set of strings chosen from the same Σ^*. In other words, a language is just a subset of Σ^* or a superset of Σ. Regular languages is the simplest class of the four Chomsky formal languages [81]. Regular expressions are used to denote the regular languages, they can represent regular languages and operate on them succinctly. The regular expression matching problem is to decide whether a given string is a member of the language defined by some particular regular expression(s). In the deep packet inspection scenario, the given string is the payload of a packet or flow.

The finite state automaton is a mathematical model of computation used to design both sequential logic circuits and computer programs. It is also a typical tool for describing regular languages, and it is equivalent with regular expressions in regular language description. More precisely speaking, for any finite state automata *M*, there exists a regular expression that describes the same language as *M*. On the other hand, for any regular expression *R*, there exists a finite state automaton that accepts the same language as *R*. There are two traditional finite state machine, named nondeterministic finite automata (NFA) and deterministic finite automata (DFA). A deterministic finite automaton M is a 5-tuple (Q,Σ,σ,q_0,F), consisting of:

1. a finite set of *states*, denoted as Q.
2. a finite set of *input symbols*, denoted as Σ.
3. a *transition function* that takes a state and an input symbol as arguments and returns a state, often denoted as σ. In the graph form of FA, σ is represented as arcs between states and the symbol labels on these arcs. If p and q are states, a is an input symbol and $\sigma(p,a)=q$, then there is an arc labeled with a from state p to state q.

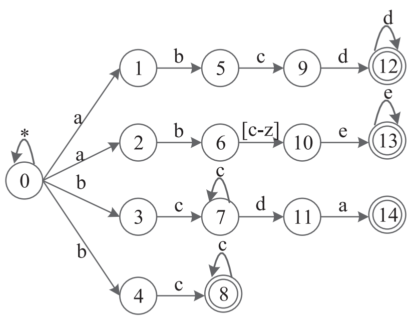


Fig. 4. A NFA accepting regular expressions of *abcd+*, *ab[c-z]e+*, *bc+da*, and *bc+*.

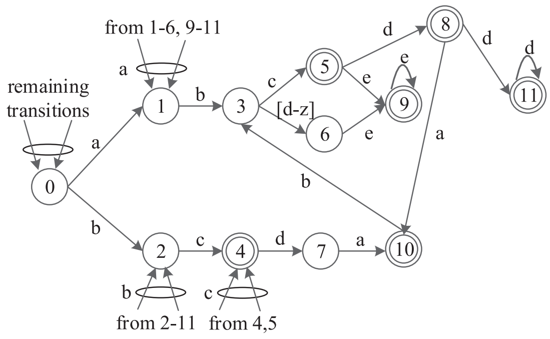


Fig. 5. A DFA accepting regular expressions of *abcd+*, *ab[c-z]e+*, *bc+da*, and *bc+*.

4. a start state in Q, denoted as q_0;
5. a set of *final* or *accepting states* F, and F is a subset of Q.

The language described by a DFA is the set of strings that it accepts. Next, we illustrate how a DFA decide to whether accept or reject a string. For a string $a_1a_2a_3 \cdots a_n$, the DFA starts from the initial state q_0 and reads the first symbol a_1. Then the σ function is invoked to get the next state, suppose $q_1 = \sigma(q_0,a_1)$, then DFA enters state q_1, this state is also called the current active state. Then, the DFA reads the next symbol a_2, and consult the next state that equals $\sigma(q_1,a_2)$, here denoted as q_2, and takes it as the new current state. The process is continuing in this manner until the end of the string, During the process, the DFA visits a sequence of states $q_1q_2q_3 \cdots q_n$, if q_n is a member of F, then the DFA accepts string $a_1a_2a_3 \cdots a_n$, otherwise the DFA rejects it. This is also a general matching process of DFA.

Analogously, NFA is also a 5-tuple (Q,Σ,σ,q_0,F), and the only difference between NFA and DFA is the *transition function* σ. For NFA, σ takes a state and an input symbol as arguments and returns a subset of Q, not a single state as in DFA. This difference also leads to the difference in matching process between NFA and DFA. For DFA, there is only one current active state as the σ returns only one state for any state and input symbol. But for NFA, as the σ returns a set of states, there maybe multiple states that are active in parallel. Despite of these differences between NFA and DFA, they can accept the same language defined by a regular expression, namely they are equivalent in regular language recognition. Next, we first discuss the general differences between NFA and DFA in detail, then introduce the generation algorithms from regular expressions to finite state automatons.

C. NFA vs DFA

To clearly illustrate this issue, we employ the examples in Becchi's dissertation [82]. The NFA and DFA accepting regular expressions of *abcd+*, *ab[c-z]e+*, *bc+da*, and *bc+* are displayed in Figure 4 and Figure 5 separately. These figures are graph forms of NFA and DFA, where numbers in circles are state IDs, arrows with symbols represent labeled transitions, state 0 is initial state and the states with double circle are final states. State transition table is another general form of finite state automata when store them in memories. It is a conventional, tabular representation of σ function that take two arguments and return one or a set of values. The rows represent state, columns represent input symbols, and the returned values indicate corresponding transition states. Attention that state transition table is more suitable for DFA rather NFA, as σ function of NFA is an irregular multi-value mapping.

The matching process takes state 0 as the initial active state, and process the input symbols byte by byte. For each step, the NFA reads one symbol and query all the destination states for current active states and symbol, and takes these states as active states for next step. A match is reported each time a final state is visited, indicating that the corresponding substring is accepted by the NFA. For example, the matching process for input text *abcda* can be represented as $(0)\xrightarrow{a}(0,1,2)\xrightarrow{b}(0,3,4,5,6)\xrightarrow{c}(0,7,\underline{8},9,10)\xrightarrow{d}(0,11,\underline{12})\xrightarrow{a}$ $(0,1,2,\underline{14})$, where the accepted states are underlined. In this example, three matches are reported, meaning that the corresponding substrings *abc*, *abcd*, and *abcda* are accepted by the regular expressions. During the process, the active state set size varies from 1 to 5, and the total number of state traversals is 21. As each state traversal involves at least one memory access, the matching for string *abcda* requires at least 21 memory accesses. In addition, active state set size also provides a measurement for memory bandwidth requirement and processing time. In some worst situations, all the NFA states maybe active simultaneously, which means that all the NFA states should be visited for the next input symbol.

For DFA, as each state has one and only one transition for each symbol in the alphabet, the size of active state set is always one during the matching process. For instance, the matching process of string *abcda* is as follows $(0)\xrightarrow{a}(1)\xrightarrow{b}(3)\xrightarrow{c}(5)\xrightarrow{d}(8)\xrightarrow{a}(10)$. As each input symbol only involves one state traversal, this leads to deterministic and limited memory bandwidth requirement. This makes DFA more attractive compared with NFA when implementing on memory-centric architectures.

On the other hand, NFA performs much better than DFA in storage requirement. In general, the number of NFA state is linear with the length of corresponding regular expression. But the conversion from NFA to DFA may bring state expansion or even state explosion, if a NFA has n states, the corresponding DFA state number can be as large as 2^n in the worst case. For example, the Snort rule "*AUTH\s[^\n]{100}*" used for detecting IMAP authentication overflow attack contains over

TABLE II
TIME AND SPACE COMPLEXITY COMPARISONS OF NFA AND DFA IN VARIOUS STRATEGIES

	Compile m regular expressions into m FAs		compile m regular expressions into an integrated FA	
	Processing complexity	Storage cost	Processing complexity	Storage cost
NFA	$O(mn^2)$	$O(mn)$	$O(mn^2)$	$O(mn)$
DFA	$O(m)$	$O(m2^n)$	$O(1)$	$O(2^{mn})$

TABLE III
THE MAPPING BETWEEN DFA STATES IN FIGURE 5 AND NFA STATES SET IN FIGURE 4

DFA state	NFA state set
0	0
1	0,1,2
2	0,3,4
3	0,3,4,5,6
4	0,7,8
5	0,7,8,9,10
6	0,10
7	0,11
8	0,11,12
9	0,13
10	0,1,2,14
11	0,12

10^{13} DFA states. When compiling multiple regular expressions together, this problem becomes more complex. Table II concludes worst case comparisons between NFA and DFA for various strategies.

For m regular expressions, there are two methods to handle them: compiling them individually into m finite automatons or compiling them into an integrated automaton [83]. For NFA, these two methods have no difference both in time complexity and space complexity. But for DFA, the integrated automaton can decrease the processing complexity from $O(m)$ to $O(1)$, which means that one memory lookup is enough to process an input symbol against all the regular expressions. While, the number of states may increase from $O(m2^n)$ to $O(2^{mn})$ in the worst case.

D. From Regular Expression to Finite State Automata

The common way to compile regular expressions involves three steps: compiling the regular expressions to NFA, converting NFA to corresponding DFA, DFA minimization.

Several algorithms [84]–[86] can be employed to generate the NFA from a given regular expression. For a set of REs, first a NFA is built for each RE, then a public initial state is added to combine these NFAs to an integrated NFA with ϵ-transition. Generally, the NFA for given regular expressions is not unique. Becchi and Crowley [87] proposed a variant of the NFA to DFA conversion algorithm to optimize the traditional NFA, and the optimized NFA can achieve a smaller state size and active state set size.

Then the integrated NFA is converted to an equivalent DFA through subset construction algorithm [88]. In essence, any DFA state represents a distinct set of NFA states which can be active at the same time. The algorithm is to explore all the possible active NFA state sets and mark them with unique identifiers, namely the DFA state ID. Table III provides the relationship between DFA states in Figure 5 and NFA states in Figure 4. It starts from the initial NFA state and keeps running until all the sets of potentially simultaneous active NFA states have been found. The following Algorithm 1 describes the classical subset construction algorithm. *Queue* and *all_subsets* are the main structures which separately maintains subsets waiting for processing and subsets that have been assigned DFA IDs. *Whileloop* and internal *forloop* are the main processes in Algorithm 1. For each loop in while, a nfa set is popped from the *queue* of unprocessed <nfaset,dfaid>. Then, in the internal for loop, a nfa set named *nfaset_c* is computed for each character c in Σ. If *nfaset_c* does not exist in *all_subsets* which record all appeared nfa sets, a new DFA id named *dfaid_c* is assigned to *nfaset_c*, and <*nfaset_c*, *dfaid_c*> is added to both queue and *all_subsets*. Subset construction is very time-consuming as each new generated NFA subset should be compared with all existed NFA subsets to determine whether to regard the current NFA subset as a new DFA state. For example, suppose we already have 999999 DFA states and corresponding NFA subsets, when a new NFA subset is generated, it should be compared with all the 999999 subsets to see whether it has already existed. In addition, each subset comparison may relate to $O(n)$ state comparison, where n is the NFA size. This is very challenging for applications like NIDS where the patterns need to be updated frequently.

The final step is DFA minimization, which transforms the given DFA into an equivalent DFA with minimum number of states. These two DFA recognize the same regular language, and the minimum DFA is unique. The minimization process is implemented by removing or merging DFA states without changing the languages it accepts. Several algorithms are published in the automata theory textbooks [86], and Hopcroft's method remains the most efficient algorithm [89]. This algorithm has worst case time complexity of $O(n \log n)$, and the average case complexity is even $O(n \log \log n)$, where n is the state number of original DFA. Although DFA minimization can decrease the state number to some extent, this linear reduction is negligible in comparison with DFA's exponential expansion.

E. Goals and Challenges

1) Goals: The DPI system should satisfy some specific objectives to support the ever-increasing link speed and pattern scale. Here, we present the design criteria [90], [91] in DPI system.

1. **Deterministic high performance:** The system should provide wire-speed processing rates of multiple tens of gigabits per second to meet the real time inspection requirement. In addition, the throughput should be independent of input stream and pattern characteristics to guarantee a deterministic performance.

Algorithm 1 Classical Subset Construction Algorithm Converting NFA to DFA

```
Require:
    NFA N = (Q, Σ, σ, $q_0$, F)
Ensure:
    DFA D = (Q′, Σ, σ′, $q'_0$, F′)
 1: Q′←∅; F′←∅; queue ← ∅; all_subsets ← ∅;
    nfaset←{$q_0$}; dfaid←0; dfa_num←1;
 2: push(queue, <nfaset, dfaid>)
 3: all_subsets←all_subsets ∪ < nfaset, dfaid >
 4: Q′←Q′∪{dfaid}
 5: while queue≠∅ do
 6:   <nfaset, dfaid> = pop(queue)
 7:   for each c∈Σ do
 8:     nfaset_c ← $\bigcup_{s \in nfaset} \sigma(s, c)$
 9:     if nfaset_c is found in all_subsets's nfaset domain then
10:       dfaid_c ←dfaid corresponding to nfaset_c in all_subsets
11:     else
12:       dfaid_c ←++dfa_num
13:       Q′ = Q′ ∪ {dfaid_c}
14:       if nfaset_c ∩ {F}≠∅ then
15:         F′ = F′ ∪ {dfaid_c}
16:       end if
17:       push(queue, <nfaset_c, dfaid_c>)
18:       add <nfaset_c, dfaid_c> to all_subsets
19:     end if
20:     σ′(dfaid, c)←dfaid_c
21:   end for
22: end while
```

2. **Scalability of patterns** The system should be able to support patterns of a considerable scale without degrading the performance substantially, and the scale and complexity of patterns depend on specific applications. For network intrusion detection system, the pattern scale maybe hundreds of thousands. This requires succinct memory structures of the corresponding automata to fulfill the memory space limitations.
3. **Dynamic update** The update performance is a crucial factor for DPI applications especially for NIDS, because the attack characteristics change rapidly, the system must react quickly.
4. **Additional functions** In practice, millions of sessions maybe opened concurrently, and these packet flows are processed in an interleaved fashion, by storing and retrieving the scanning state for each session. To guarantee the wire-speed throughput, small session state is required to minimize the context switching overhead. In addition, the system should enable the user to customize specific subset of patterns to inspect, as in Snort.

2) Challenges: The challenges mainly originate from the issues between the hardware resource limitations and the ever-increasing link speed and pattern scale. As DFA has a relatively high matching efficiency, most researches apply DFA for regular expression matching on memory based architectures. In practice, DFA is organized as a two-dimensional matrix and stored in the memory. Each time the matching engine reads an input character, then inquires the matrix once for the next active state through simple address calculation based on the current active state and the input character. As the engine's calculation speed is always faster than memory access, thus for a given input sequence, the processing time is mainly determined by the memory access latency. Thus for a better performance, the DFA should be stored in faster memories. However, most fast memories have very small capacity, usually not more than 10 MB. This capacity can only hold an original DFA with 20k states, which is much smaller than a modern DFA.

On the other hand, with the rapid growth of patterns in various applications, the size of DFA expands in a squared or even exponential speed. For example, the number of rules in Snort has been more than 10 thousands, which further widens the capacity gap between the space of fast memories and that the DFA memory requirements. As massive redundancy exists in state transition table of DFA, a natural method is to compress the table to fulfill the memory space limitations. The resulting DFA-like FSM structure is more compact and irregular than that of DFA, thus demands more memory accesses for processing a symbol. A responsible compression algorithm should consider the memory bandwidth requirement as well as compression ratio, early researches mainly focused on this direction. Compilation time is another factor which should be concerned because each time a new NFA subset is built the subset should be contrasted with all the existing subsets.

Traditional compression methods could achieve compression ratios over 90% on the basis of DFA, in some cases however, the explosion even makes the DFA construction infeasible under the current general computing capabilities. State explosion is the core issue in regular expression matching (REM), neither memory requirement nor matching throughput will be a problem without state explosion. To restrain state explosion, many efforts have been made from several directions as classified in Section IV-C. These novel FSMs are called scalable FAs which avoid state explosion with novel compact structures. However, matching efficiency remains a problem for these FAs in practice. In a nutshell, the growth in scale and complexity of patterns bring a tremendous challenge for the employment of DFA. For practical implementations, a sophisticated FSM must be scalable to support a large set of patterns. Table IV summarizes FSM solutions and memory/efficiency results with the increasing pattern scale and complexity. In the next section, we first analyze reasons for state explosion from perspectives of both pattern features and NFA relationships, then discuss and category the DFA based compression algorithms and scalable FAs thoroughly.

On the other hand, the existing link speed of backbone Internet has reached up to 40 Gbps (OC-768) with the rapid spread of various applications such as big data, HD video, cloud computing. And the link speed of current fast Ethernet has even risen to 100 Gbps, traditional architectures are unbearable to support for the ever-increasing speed [92]. As mentioned above, the performance is mainly decided by

TABLE IV
FSM SOLUTIONS AND MEMROY/EFFICIENCY RESULTS

Problem Scale	DFA States	Memory Cost	Solutions	Performance
Simple	10K	10MB	No need	Tens of Gbps
Medium	100K	100MB	Compression	about 10 Gbps
Large	1M	1GB	Compression or scalable FAs	about 1 Gbps
Huge	Infeasible		Scalable FAs	about 0.1 Gbps

memory access latency. Even with the SRAM of 250 MHz frequency and 3 cycles latency, a single DFA engine can only reach 0.67 Gbps, which is two orders of magnitude lower than requirement. Hence, the throughput is a great challenge for DPI systems. As parallel platforms, such as FPGA, GPU and TCAM, are widely used in modern network devices, many implementations employed these platforms for pattern matching. Each processing unit is regarded as a matching engine, and multiple streams are processed separately and concurrently on these units, resulting a linear speedup in throughput. Issues about matching acceleration on parallel platforms will be presented in Section V.

IV. AUTOMATA OPTIMIZATIONS

Pattern matching is a core and critical step, as well as a bottleneck, in DPI applications. In the past years, substantial efforts have been put in optimizing automaton-based pattern matching in order to improve the overall performance of DPI. This section serves a summary on what have been done in this aspect including recent results. As we discussed, state expansion or explosion is the primary obstacle for implementing DFA based regular expression matching. We first analyze the reasons of state explosion from the aspects of pattern characteristics and the semantic relationships among NFA states. Then, the classical and significant DFA compressing algorithms are discussed and categorized, including the novel researches published in recent few years. These algorithms can reduce memory storage requirement only on the basis of DFA, but for some scenarios of large or complex rule sets the DFA generation is even infeasible. Another kind of automatons can fix this problem, and we named them as scalable FAs, indicating that they are able to support complex and large scale pattern sets. In addition, we also classify them into different categories as Grouping FAs, semi-determined FAs and Decomposed FAs in detail. Finally, we also provide the readers with suggestions of automata optimizations in different scenarios.

A. Analyzing State Explosion

With the ever-increasing scale and complexity of pattern sets, there is also an exponential growth in the number of states if we want to construct a DFA to capture all these patterns. This explosion even makes it infeasible to construct a DFA under the current general computing capabilities. We have tried to compile half (57 regular expressions) of the rules in L7-filter with Becchi's compiler called RegEx Processor [93] in our server, the program kept on running even when the DFA state number has exceeded 50 million, for which the space cost was already more than 50 GB. This storage requirement exceeds far beyond most current network equipments. We will try to provide a detailed description on the reasons for the state explosion from two perspectives: pattern features and NFA relationships.

TABLE V
RAPID STATE INFLATION FOR TYPICAL PATTERNS

Pattern features	Examples	Space complexity
Pattern with start anchor '^', a character class with length restriction k, where the character class overlaps with the prefix	$\hat{}A+[A-Z]\{k\}B$	$O(k^2)$
Pattern starting with ".*", a character class with length restriction k, where the character class overlaps with the prefix	$.*AB[A-Z]\{k\}C$	$O(2^k)$

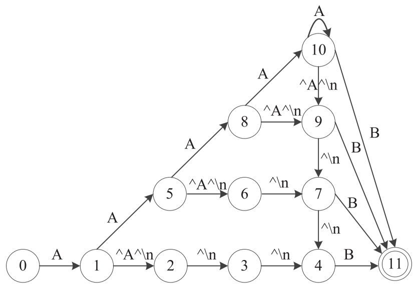


Fig. 6. DFA for expression "^A+[^\n]{3}B."

1) Analyzing State Inflation From Perspective of Pattern Features: Some studies [94], [95] investigated the causes of state explosion and proposed instructions to avoid or solve it. State explosion generates from the ambiguity of metacharacters in regular expressions because massive states are needed to record all possible input sequences. While not all combinations of metacharacters can cause state explosion, Yu *et al.* [94] summarized some special models which could lead to rapid state inflation. Generally, there are two kinds of inflation, inflation when compiling a single expression and inflation when compiling multiple expressions together.

The former inflation mainly rises from the combination of length restrictions with wildcards such as '.' and large character classes which can cause an inflation of polynomial magnitude or even exponential magnitude. Table V shows the typical pattern structures which can cause rapid state inflation in a single rule.

When an expression starts with a start anchor '^' and contains a wildcard with a length restriction k, if the wildcard has overlaps with the prefix, the expression will have an inflation of $O(k^2)$. Take the expression "^A+[^\n]{3}B" as an example, the DFA is shown in figure 6. The character class "[^\n]" overlaps with the prefix 'A', hence, the input character 'A' can be matched as part of the prefix "A+" as well as the part of "[^\n]{3}". To keep the matching correctness, the DFA must be able to record all the possibilities. In this example, states from 1 to 10 are used to omit the ambiguity in the expression. For an expression with length restriction of k of this type, k^2 states are needed to record all the matching paths.

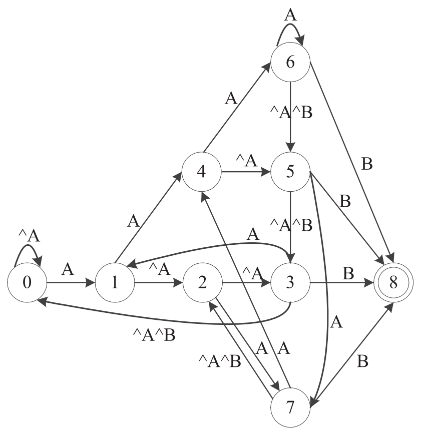


Fig. 7. DFA for pattern "$.*A.\{2\}B$."

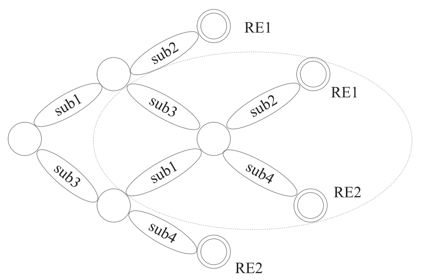


Fig. 8. DFA structure for "$sub1.*sub2$" and "$sub3.*sub4$."

Despite the above polynomial inflation, a more rapid inflation of exponential size arises if the expression starts with "$.*$". In this type, the pattern also has length restriction for the wildcard which overlaps with the prefix. A DFA example for the expression "$.*A.\{2\}B$" is shown in figure 7. This kind of patterns have more ambiguity because in any step of the matching process, the occurrence of 'A' can represent the start for a new matching instance. Thus, any state should have individual transitions for input 'A' which resulting more intermediate states. For a pattern of this type with a length restriction of k, $O(2^k)$ states are required to omit the ambiguity.

In practice, a more popular situation for state explosion occurs when compiling multiple patterns into a single DFA even each of these patterns has no state inflation. This kind of explosion is mainly caused by "$.*$" inside the patterns [95] which can represent any input sequence. When compiling multiple patterns together, each "$.*$" in a certain pattern will generate duplications for all the other patterns. Suppose that we have two patterns expressed as "$sub1.*sub2$" and "$sub3.*sub4$", each "$subi$" represents a sub-pattern and the integrated DFA structure is shown in figure 8. In this figure, the ellipses denote FAs for the sub-patterns. As we can see, "$.*$" in pattern 1 duplicates the FA for pattern 2, namely the FA for "$sub3.*sub4$" inside the dotted circle. Similarly, "$.*$" in pattern 2 generates the FA for pattern 1 inside the dotted circle. In practice, most of the patterns contain multiple "$.*$", each "$.*$" in a pattern will duplicate FAs for all the other patterns. Hence, the interaction among these patterns causes the state inflation much more rapid.

2) Analyzing State Inflation From Perspective of NFA Relationships: Recent studies [96], [97] have exploited the origin of state inflation from the perspective of relationships of NFA states. First, we present the relationships of NFA states and the corresponding DFA states by semantics. Then, we define the relationships of NFA states and list all three kinds of relationships. Third, we discuss which kind of relationship could cause state inflation and how much expansion it causes.

An NFA is often formalized as a quintuple M=(Q,Σ,σ,q_0,F), where Q is the set of finite states, Σ is the input alphabet, σ is the transition function, $q_0 \in$Q is the start state, and F$\subseteq$Q is a set of accepting states. As for a current NFA state s and an input symbol c, σ may have multiple destinations, meaning a set of next NFA states. Subset algorithm is to search all such NFA sets, within which the NFA states could be active concurrently, and each NFA set is regarded as a DFA state. In other words, a DFA state represents a set of corresponding NFA states which can be activated concurrently by some input sequences.

In a NFA state transition graph, there are many paths from the initial state q_0 to a given state q. Each path is corresponding to an identical input sequence, and all these sequences are languages recognized by state q, denoted as $L(q)$. The relationship between two NFA states means the relationship of their corresponding language sets, which can be summarized as *exclusive*, *conflict*, *inclusive* according to [97].

1) Exclusive: Two NFA states p and q are exclusive means there is no interaction between $L(p)$ and $L(q)$, there exists no string s satisfying that after traversing s, p and q are both active.
2) Conflict: The conflict relationship should meet three conditions, for NFA state p and q, $L(p) \cap L(q) \neq \emptyset$, $L(p) \not\subset L(q)$, and $L(q) \not\subset L(p)$, denoting that they can identify some same strings but neither state can be represented by the other.
3) Inclusive: States p includes q means that $L(q) \subset L(p)$, denoting any string activating state q will also activate state p.

All NFA relationships have been listed above, next we quantify state inflation in DFA generation according these relationships. As has been discussed, each DFA state represents a set of NFA states which can be active concurrently, and different DFA states represents different combination of NFA states. Thus, we only need to consider which kind of NFA state relationship could introduce concurrently active NFA state set. If any two NFA states are exclusive, no new subset will be generated during the NFA to DFA conversion, which means that each NFA state represents a DFA state and the state number is the same after conversion. If any two NFA states are inclusive, all the languages sets for all NFA states must have the relationship of totally ordered set relation. The language set identified by any combination of NFA states equals to the smallest language set identified by these states, implying that no combination of NFA states can identify a new language set. Thus, no state inflation occurs.

The only exception is the conflict relationship. For a NFA with conflict relationship for any pair of NFA states, if the number of NFA state |Q| = 2, the corresponding DFA has three states, representing as NFA state set of $\{q_{n1}\}$, $\{q_{n2}\}$ and $\{q_{n1}, q_{n2}\}$. When increasing |Q| to 3, at least three more DFA states will be generated, denoted as $\{q_{n3}\}$, $\{q_{n1}, q_{n3}\}$ and $\{q_{n2}, q_{n3}\}$. If q_{n3} is conflict with $\{q_{n1}, q_{n2}\}$, an additional state $\{q_{n1}, q_{n2}, q_{n3}\}$ will be appended. In general, by adding a new NFA state which is conflict with all existing NFA states, the number of corresponding DFA state will increase at least |Q|+1 and at most |Q|+1+|D|+1, where |D| is the corresponding DFA state number before adding the new NFA state. According to this law, a NFA with conflict relationship for any pair of NFA states will produce at least $3+(2+1)+(3+1)+\ldots+|Q| = |Q||Q+1|/2$ DFA states and at most $3+(3+1)+(7+1)+\ldots+2^{|Q|-1} = 2^{|Q|}-1$ DFA states. This result is consistent with Yu's analysis in Table V.

B. State Transition Table Compression Algorithms

As DFA has an excellent matching performance with naturally high space complexity, most software methods focused on DFA state transition table (STT) compression [51], [79], [87], [98]–[110]. The STT is a kind of standard representation for FSM. It is a two-dimensional matrix, the rows represent DFA states and the columns represent input characters. For an ASCII alphabet, the number of column is 256. The element in row *i* and column *j* indicates the next transition state when a current state *i* receives an input *j*. Therefore, the space requirement of a DFA with *n* states is $n * |\Sigma|$. While in practice, the STT has massive redundancies from both intra-state transitions and inter-state transitions.

A sophisticated compression algorithm can achieve a reduction ratio of more than 95%. The nature of compression is to trade off time for space, leading to an inevitable sacrifice in performance. Therefore, a responsible compression algorithm must consider the memory lookup requirements as well as compression ratio. The current compression algorithms are divided into three categories: state merging [98], alphabet re-encoding [87], [99], [105], [106], and transition compressing [51], [79], [98], [100]–[104], [106]–[110]. State merging and alphabet re-encoding can reduce the number of STT from state dimension and input dimension respectively, while transition compression replaces redundant transitions with compact representations. As state merging and alphabet re-encoding only work well for small scale DFAs, most researches focus on compression of transitions.

1) State Merging Algorithms: Becchi and Cadambi [98] proposed a state merging algorithm which merges states with some same destinations into one state. State merging reduces the row number of the transition table. For simplicity, we employ an example to clarify the main idea.

Figure 9(a) shows a simple DFA, we omit transitions leading to state 0 for convenience. In the state merging algorithm, two states can be merged if they have one or more destination(s) that are the same no matter whether the destinations have the same input labels. For example, state 3 and state 4 have a destination to state 5, even the labels *e* and *f* are not the same, they can still be merged into one state as shown in Figure 9(b). To maintain the validity, the author employed labels to distinguish the original states where label 0 represents

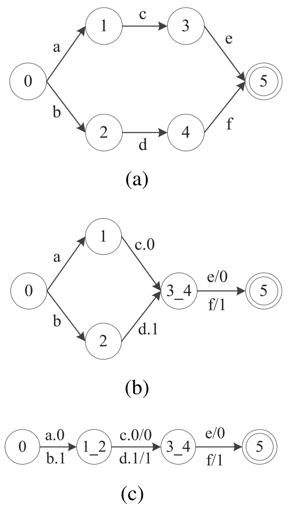


Fig. 9. A simple example for state merging.

state 3 and label 1 represents state 4. Further, transition *c*.0 means that state 1 transfers to state 3_4 with a label of 0 when receiving character c while transition *e*/0 means that state 3_4 with a label of 0 transfers to state 5 when receiving character *e*. As we can see, the merging of state 3 and state 4 creates another merging opportunity for state 1 and state 2, and they can be further merged into state 1_2 as shown in Figure 9(c).

As the requirement for merging is very low and dynamic merging creates more merging opportunities, the algorithm can get a memory reduction of 90%. However, to achieve the best compression, the algorithm needs to compute the weights between any two states in every merging iteration, which results in a computational complexity of $O(n^3 \log(n))$ where *n* is the number of the original DFA states. In addition, the data structure of compressed DFA requires three times of memory accesses in processing an input character.

2) Alphabet Re-Encoding Algorithms: It is observed that in most cases, some input characters are equivalent which means that for any given state *s*, the next destinations for these characters are always same. To reduce the columns of transition matrix, alphabet re-encoding [87], [99], [105], [106] merges equivalent characters and re-encodes the input alphabet. Taking figure 10 as an example, figure 10(a) represents the STT of an original DFA and figure 10(b) indicates the DFA after alphabet re-encoding. As we can see, in figure 10(a) for any given state *s*, $\sigma(s, a) = \sigma(s, b)$ and $\sigma(s, d) = \sigma(s, e)$, which means that *a* is equivalent with *b* and *d* is equivalent with *e*. Therefore, the alphabet can be encoded as figure 10(b), where equivalent characters are merged and re-encoded. For correct addressing, each input symbol should be mapped to the equivalent character class through a mapping table. Kong *et al.* [105] observed that, in many cases a set of characters may behave equivalently for the vast majority of states and only a few states behave individually on these characters, which limits the further character merging. For a more compact transition table, Kong *et al.* [105] divided the entire states into multiple subsets without overlapping and generated an individual mapping table for each subset.

	input				
state	a	b	c	d	e
A	B	B	D	C	C
B	C	C	C	D	D
C	D	D	A	B	B
D	A	A	A	A	A

(a)

	input		
state	0	1	2
A	B	D	C
B	C	C	D
C	D	A	B
D	A	A	A

(b)

Fig. 10. Example for alphabet re-encoding.

Alphabet re-encoding can get a space reduction linear with alphabet reduction, which is 40% in this example. However, in the new DFA, the input characters must be mapped to the new alphabet before matching, which will increase the processing procedures. To decrease additional mapping cost, Brodie *et al.* [99] devised a hardware circuit called ECI to perform the conversion in one cycle. Becchi and Crowley [106] also proposed to cache the mapping table and pipeline the access to accelerate the mapping procedure.

3) Algorithms of Transition Compression: The previous methods perform compression through reducing the numbers of rows or columns of the original STT, while in this subsection, compression methods mainly exploit redundancy among transitions. Due to the powerful expressive ability of regular expression, a regular expression may represent thousands of strings and the DFA must track any possible input sequence. Hence, massive redundancy exists among the transitions. There are two kinds of transition redundancy: intra-state redundancy and inter-state redundancy. The former redundancy indicates that for a given state, it has only a few distinct destinations, which is much less than the alphabet number $|\Sigma|$. Meanwhile, there are massive equivalent transitions among different states, especially in neighbor states. Equivalent transitions between states means that these states have same destinations for some identical characters.

The former redundancy can be effectively addressed by bitmap encoding [51]. It is observed that, for a given state, most of its transitions point to a same destination, especially to the initial state, and this special destination is called default transition. Tuck proposed a basic bitmap encoding [51] to omit these default transitions. Instead of recording destinations of every alphabet character, bitmap encoding only stored undefault transitions and the bitmap was employed to lookup these transitions. Suppose a state has transitions as figure 11(a), then we can get a bitmap representation in figure 11(b). In figure 11(b), the bit in the bitmap indicates whether the transition is undefault for a given character and the compact transition table only stores undefault transitions. For an undefault transition, the destination can be indexed by adding the number of 1*s* before that bit to the base transition number. With the traditional transition matrix, the 256 transitions need 1024 bytes while the bitmap encoding only needs 56 bytes.

However, as the existence of same destinations, there is still some redundancy in the transition table. As in figure 11(b), both of state 1 and state 5 are the same destinations for multiple transitions. Becchi and Cadambi [98] introduced an indirect pointer table to solve this problem. In this scheme, the

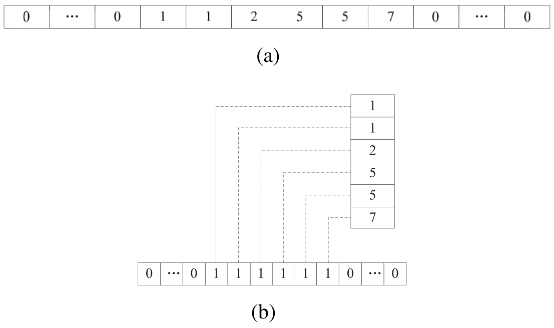


Fig. 11. A simple example for bitmap encoding.

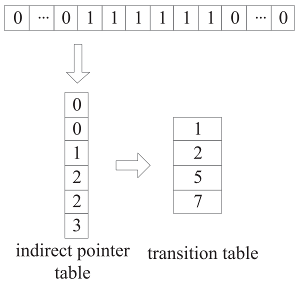


Fig. 12. Bitmap encoding with indirect pointer table for figure 11.

indirect pointer table is employed as the index to the transition table and the transition table only needs to store the distinct destinations. Figure 12 shows the example for figure 11(b). In figure 12, the bitmap indicates addresses to the pointer table, then the corresponding pointers can be used to index the destination states in the transition table. As the alphabet size is much bigger than the undefault transitions, the space cost of indirect pointer table can be very small. In figure 12, the space for pointer table is 12 bits and the total space cost is 396 bits. In practice, the space can be much smaller than bitmap encoding while the cost is one more memory access for each input character.

The main disadvantage for bitmap is the counting of 1 before a certain position and multiple memory accesses for a long bitmap. While this issue can be resolved by dividing the long bitmap into multiple short bitmaps and appending each bitmap with additional bits indicating the number of 1*s* before it. Thus for each input symbol, only a short bitmap need to be accessed and the counting is confined in this short bitmap. Similar with bitmap encoding, Becchi and Crowley [106] proposed indirect addressing for intra-state redundancy. In indirect addressing, a state identifier is represented with labeled transitions information, which can allow a single memory access for every input symbol at the cost of a space-cost encoding and time-cost hash computation.

Further, Qi *et al.* [110] proposed a bitmap-based two-dimensional DFA compression algorithm, which can achieve high compression from both intra-state transitions and inter-state transitions. For intra-state compression, the algorithm uses an advanced bitmap technique which can supports fast bitmap computation. For inter-state compression, it employs a 2-stage grouping algorithm which efficiently compress inter-state redundancy meanwhile maintaining the intra-state position information for fast lookup. In addition, the author also devised a hardware lookup engine for high-performance lookup.

Antonello *et al.* [79] proposed another novel compression algorithm called ranged compressed deterministic finite automaton (RCDFA) to reduce the intra-state redundancy without additional memory lookups. The author observed that there exists large consecutive transitions going to the same destination state for many states. Rather than using bitmaps, RCDFA represents these consecutive transitions as a unique ranged transition, thus involves no additional memory lookups. Experimental result shows that RCDFA can achieve 97% compression ratio over the original DFA. Further, with three optimizations and advanced layouts, a enhanced RCDFA [109] is proposed to achieve additional compression or decrease the average number of characters inspected. The author observed that, there is no need to report all the appearance of a pattern as the leftmost matched instance suffices. By relaxing this requirement, three optimizations namely removing nonforward states, removing acceptance state's transitions and removing states reachable only from acceptance states are equipped with RCDFA. In addition, three encoding algorithms named Relative Linear Encoding, Adaptive Linear Encoding and Hard-Coded Encoding are also presented for representing (compressed) DFAs. With these optimizations, the advanced RCDFA can reach a high and steady compression ratio of around 98% and 32 times performance on average.

Many compression algorithms [100]–[104], [106], [107] focused on the equivalent transitions between different states and a majority of them were based on the research of Kumar *et al.* [100]. Kumar *et al*. proposed a new representation D^2FA to omit equivalent transitions among different states. Take the transition table in Figure 13 as an example, most states have the same destination for the identical input and the only exception is input *c* for state 2. In D^2FA, for a group of states with equivalent transitions, it only stores equivalent transitions once in a state and adds a default transition to this state for the other states. Figure 14 shows the D^2FA representation for Figure 13. In matching process, if there is no destination for a given character, the current state follows the default destination without consuming the character until a labeled transition appears for the input character. As there is only one default transition for each state, it's important to choose appropriate default transitions to achieve largest compression. Kumar *et al*. [100] employed algorithms for maximum weight spanning tree problem of undirected graph and picked some states as roots to determine the directions of default transitions. In practice, D^2FA can get a significant compression ratio of 95% at the cost of multiple memory accesses for a character. As there is no limitations for default transitions, the default transition paths can be very long, meaning frequent memory accesses for an input character. To guarantee the worst case performance, a bounded diameter for the spanning tree was proposed to limit the length of default transitions, which made a tradeoff between compression ratio and worst case performance. However, constructing such a maximum spanning tree is NP-hard, and there is no guarantee to generate a smallest D^2FA.

	input			
state	a	b	c	d
0	1	2	0	3
1	1	2	0	3
2	1	2	4	3
3	1	2	0	3
4	1	2	0	3

Fig. 13. Transition table with massive equivalent transitions.

	input				
state	a	b	c	d	default
0	1	2	0	3	
1					0
2			4		0
3					0
4					0

Fig. 14. D^2FA representation for figure 13.

Furthermore, to avoid exhaustively default traversal in D^2FA, Kumar *et al.* [101] proposed an improved content addressed D^2FA (CD^2FA). In CD^2FA, each state is associated with a content label, which contains transition information indicating which set of characters have labeled transitions for the states along the default transition path. For a given input character without defined transition on the current state, a hash function directly locates the first state in the default path which has a labeled transition for the character via content label. With this content label, any input character can be processed in one memory access by skipping the intermediate states in default path. As the content label is very small, usually 32 bits or 64 bits, it remarkably limits the bounded diameter for the spanning tree which will have a negative impact on compression ratio.

Based on D^2FA, Becchi and Crowley [102] proposed a simple but effective method called backwards labeled transitions to overcome the inefficient throughput in D^2FA. All states are attached with a depth label which indicates the minimum number of states to be visited from the initial state to the states. The backwards labeled transitions only allow default transitions from big-depth states to small-depth states. This straightforward limitation achieves a significant improvement in matching throughput, which requires not more than $2N$ memory accesses for an input sequence with length of N. In this mechanism, any character processing involves a label transition and the depth increment is not more than 1. For a string with length of N, the depth increase for all characters will not exceed N. Meanwhile, as default transition only goes backward, any default transition will cause the depth of the current state to decrease at least 1. Hence, the number of total default transitions during processing will not exceed N. Essentially, the limitation for the direction of default transitions from big-depth states to small-depth states is to purposefully choose the states with smaller depth as the roots. It is mainly based on the observations that most packets match none of the rules and most transitions will go back to the initial state or their neighbors during processing. As states with smaller depth are more likely to be visited, choosing them as roots of default transitions can avoid massive unneeded transitions. Further, Becchi and Crowley [106] extended this method by limiting that the depth difference should be more than k for any default transitions, which leads to a memory access of $N(k+1)/k$ for an input sequence with length of N. With this extension, the balance of storage requirement and memory bandwidth of D^2FA can be tuned with the factor of k. Experimental results show that, compared with D^2FA, A-DFA can achieve comparable compression ratio with lower memory bandwidth requirements or greater compression ratio for a given memory bandwidth limitation.

As D^2FA and above refined automatons are constructed on the basis of DFA, the generation of DFA is a remaining problem when state explosion results in an infeasible DFA. To avoid state explosion in DFA generation, Patel *et al.* [107] proposed a "Minimize the Union" framework which constructs separate D^2FAs for each pattern and merges multiple D^2FAs into a final D^2FA. As there is no need to generate an integrated DFA aforehand, this mechanism appears much more space- and time-efficient. Experimental results show that this method performs an average of 1500 times less memory and 155 times faster compared with traditional D^2FA. They can even construct a D^2FA with over 80000000 DFA states consumes only 1 GB memory space and 77 mins.

Considering that D^2FA only achieves transition sharing among states, Liu and Torng [108] proposed an overlay automata called OD^2FA to achieve further state sharing by merging multiple DFA states originating from the same NFA state into a super-state. Each regular expression is compiled to an individual OD^2FA, then all individual OD^2FAs are merged into an integrated OD^2FA, thus to avoid the building of the integrated DFA.

Ficara *et al.* [103] proposed another compact representation called σFA which mainly focused on the massive redundancy between adjacent states. For any state, σFA only stores the transitions which are different with its parents. The parent relationship is defined as follows: if state s has any transition to state t, then state s is a parent of state t. A σFA transition table for DFA in figure 13 can be represented as in figure 15. Before matching, all the transitions of initial state is saved in a local transition set in local memory which represents the transitions for the current state. The matching process is continuously getting the next state from local transition set and updating it with the transitions of the next active state. The update of local transition set only involves transitions which are different with the previous active state and the number of different transitions between adjacent states is around 10 in practice. Further, a *char − state compression* is proposed to reduce the bits to represent the addresses of states in slow off-chip memory. The direct benefit is that the update operation can be completed in one memory access to the off-chip memory. As the computation and operations on local memory can be negligible compared with memory access to the off-chip memory, the performance can be approximated as one memory access per character.

	input			
state	a	b	c	d
0	1	2	0	3
1			0	
2			4	
3			0	
4			0	

Fig. 15. σFA representation for figure 13.

TABLE VI
COMPARISONS AMONG CLASSICAL TRANSITION COMPRESSION ALGORITHMS

Algorithms	Compre-ssion ratio	Time complexity	Space complexity	memory bandwidth
D^2FA	95%	$O(n^2\log(n))$	$O(n^2)$	no guarantee
A-DFA	>90%	$O(n^2)$	$O(n)$	$\leqslant 2$
σFA	>90%	$O(n\times\|\Sigma\|^2)$	$O(n\times\|\Sigma\|)$	≈ 1

σFA combines compression with matching process, thus, there is no need to additionally save the redundant information. While for a given character, a state need to store the transition as long as the transition is different any of its parents' transitions. In figure 15, the σFA stores transitions of character c for state 1, 3, 4 as state 2 has the different transition for character c and state 2 is parents of these states. Thus compression of σFA is less effective than D^2FA. Further, Ficara proposed an improved σFA called $\sigma^N FA$ [104] which can remove the compression limitations while maintaining the same performance as σFA.

Table VI shows a comparison among some of above proposals. Except for compression ratio and memory bandwidth requirements, time complexity and space complexity are also selected as comparison metrics. In general, compared with transition compression methods, state merging algorithms have a much higher time complexity of $O(n^3\log(n))$ and more memory access requirements to process each symbol. Similarly, alphabet re-encoding algorithms only work well for small-scale DFAs. Thus, transition compression are more widely used in practice and we only make comparisons for these methods. D^2FA has the best compression ratio but the memory access number is uncertain as no limitation has made to default transitions. A-DFA achieves a certain upper bound for memory bandwidth by limiting the direction of default transitions, which will sacrifice a little compression ratio. With the assistance of *char − state compression*, σFA performs well both in compression ratio and memory bandwidth. As the metrics of compression ratio and memory bandwidth are associated with regular expression sets and input streams, a fair and accurate comparison should be made under same conditions for engaged readers, and we only list the average performances here.

C. Scalable FAs

An increasing number of researches [90], [94]–[97], [111]–[118] have focused on state explosion and many new kinds of matching automata have been designed to support large scale pattern matching. Unlike traditional DFAs or compressed DFAs discussed in the previous subsection, these automata can compile hundreds to thousands of patterns to an integrated FA without state explosion, which are very useful when DFA is infeasible for large complex patterns. These automata are more NFA-like, and we call these automata scalable FAs to distinguish them from traditional DFA or DFA-like automata which cannot solve state explosion. In addition, not like the compressing methods, scalable FAs are not constructed on the basis of DFA. Besides avoiding state explosion, these automata must also consider about the matching performance. Hence, most scalable FAs are combined with special hardware architectures to accelerate the matching process. In this subsection we will introduce the scalable FAs, and the hardware architectures will be discussed in Section V.

We classify the scalable FAs into the following catalogs, named multi-FA based on rule grouping [90], [94], [97], [111], semi-determined FA [95], [96], [112], [116], FA based on rule decomposition [112]–[115], [117]–[119], etc.

1) Multi-FA Based on Rule Grouping: As analyzed previously, compiling multiple patterns into a single DFA may cause state explosion even when all the patterns have linear state numbers when compiled separately, and this phenomenon is called pattern interaction. Rule grouping [90], [94], [97], [111] is a nature method to group the rules into multiple groups and generate a DFA for each group to avoid state explosion. As different combinations of patterns have different magnitude of interactions, the goal of rule grouping is to divide the patterns interacting seriously into different groups and leaving the patterns in the same group with least interaction. The hardware foundation of multi-FA is multiple processing units such as multi-core processors, GPUs, and each unit is responsible for a group of patterns. The DFAs are stored in the local memories of these units, and a stream is processed on all the units concurrently byte by byte resulting a time complexity of $O(1)$.

To perform the grouping process, Yu *et al.* [94] proposed a greedy heuristic algorithm. Firstly, the algorithm computes whether the given arbitrary pair of rules have interaction, which means that the number of states when compiling the pair of rules together is bigger than the sum of states when compiling them separately. Then the results are used to construct a graph which indicates the interaction relationships between each pair of rules, in which each vertex represents a rule and the edge between two states means the interaction between two rules. Every time when selecting a new rule group, the algorithm selects a rule which has the least interaction with all the other rules, then, it chooses the next rule which has the least interaction with the rules in the new group one by one until the memory requirement exceeds the storage in a single matching engine. van Lunteren [90] discussed the grouping method for exact string rules, which has similar ideas with [94]. Further, Rohrer *et al.* [111] quantified the rule grouping based on [94]. Yu *et al.* [94] only defined whether two rules have interaction while Rohrer *et al.* [111] also computed how much they interact and the quantization brought an optimal or approximate optimal grouping. However, the quantization for rules' interaction relies on the construction of these DFAs, where the subset algorithm is extremely time-cost.

Previous grouping algorithms all rely on DFA generation, thus consume much time and memory space. Liu *et al.* [97] proposed an algorithm of DFA estimator, which can quickly estimate the DFA size of a given regular set without generating the DFA. Based on the DFA size estimation, a grouping algorithm of RegexGrouper was devised to implement fast grouping. As DFA generation is saved in this method, it performs much better in time and space consumption. Take the rules of L7-filter for example, RegexGrouper groups them into 7 DFAs with total states of 15578 with 3.2 minutes, compared with prior art of 279.3 minutes and 29047 states.

Though multi-FA can relieve state inflation to a certain extent, a stream must be matched with all the FAs to get the final result, which results the linear decline in performance with the number of groups increasing. In addition, rule grouping methods have no effects on single rule's state inflation. In recent studies, rule grouping is mainly used as auxiliary methods in pattern matching.

2) Semi-Determined FA: The determination from NFA to DFA brings a time complexity of $O(1)$ as well as the state explosion. However, most of the DFA states have never or rarely been accessed during the matching process. Therefore, some studies [95], [96], [112], [116] considered controlling the state explosion by adjusting the degree of determination. These FAs combine the benefits of NFA and DFA, and we call this kind of FA semi-determined FA.

To solve the problem of state explosion, Becchi and Crowley [95] proposed a hybrid FA in [95]. As discussed, the explosion is mainly caused by the "$dot - star$" conditions and character class with counting constraints. To restrain the expansion of DFA states, the subset construction stops at the NFA states which correspond to the state of "$dot - star$" constraint or the first state of character class with counting constraint. The construction result is a hybrid FA constituted of a head DFA followed by multiple independent NFAs, and the special states connecting the head DFA and tail NFAs are called border states. Figure 16(a) shows the structure of a hybrid FA. The hybrid FA has some good features: 1) the matching process starts with a DFA state; 2) tail NFA states will keep inactive until the border states are accessed; 3) there are no transitions from tail NFA states to head DFA states. During matching process, there is always an active DFA state and the tail NFA will be activated each time the corresponding border state is reached. Certainly, the explosion can be solved by stopping the subset construction once meeting the special NFA states which may cause state explosion. As the tail NFA may be activated repeatedly, it is still a tradeoff between memory occupancy and the memory bandwidth in nature. To further reduce the worst case in tail NFA, Becchi also made some improvements. For tail NFAs introduced by "$dot - star$" constraints, they can be further converted to tail DFAs through subset construction, as shown in figure 16(b). For tail NFAs introduced by counting constraints, extra counters are imported to replace the massive NFA states used for counting. In addition, the tradeoff can be adjusted by controlling the size of head DFA.

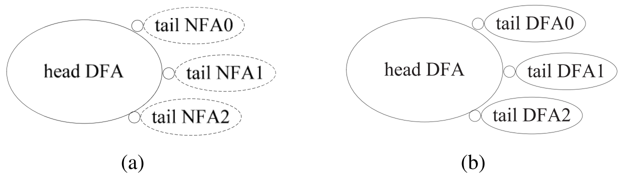


Fig. 16. Hybrid FA example.

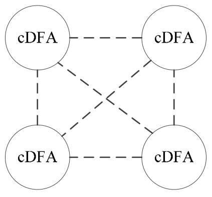


Fig. 17. Structure for an exemplified *SFA*.

To achieve expected results, the rules for matching must satisfy some features. For example, for a long complex pattern, the counter constraint or "$dot - star$" should not exist in the head or front of the pattern, otherwise, the corresponding hybrid FA will have a long complex NFA or the pattern will make a big contribution to state expansion. Kumar *et al.* [112] proposed another similar hybrid scheme, which split the patterns into frequently visited prefixes and rarely visited suffixes, and the prefixes are compiled as a DFA for fast pattern matching. The main objective is to generate a simple *prefix*-DFA, which can process most streams without activating the *suffix*-FAs. Furthermore, the splitting position is determined by computing the activation probabilities of different NFA states with training traces, which is more purposeful than Becchi's method.

Above two mechanisms both exploited hybrid FAs with a fast *prefix*-DFA and multiple associating slow *suffix*-NFAs with the assumption that most benign streams can only visit a small prefix portion of patterns in NIDS. However, it is vulnerable as attackers can easily construct malicious packets which activate the *suffix*-NFAs quickly and remain activation among these *suffix*-NFAs. Yang proposed another kind of novel semi-determined FA named *SFA* in [96]. The author proved that state explosion is caused by conflicting NFA states. The goal of *SFA* is to separate these conflicting states into p constituents, each constituent is compiled to a DFA which is called *cDFA* and transitions exist among different *cDFAs*. Figure 17 shows an exemplified *SFA* structure. These p *cDFAs* work in parallel and each of them has an initial state. Upon receiving an input character, each *cDFA* will take a transition inside itself and at most one transition to each of other $p - 1$ *cDFAs*, which means that each *cDFA* may have p next active states when processing a character. Yang defined priority for states in the same *cDFA*, hence, there is only one active state anytime in a *cDFA*. The transitions inside a *cDFA* is deterministic and transitions among *cDFAs* are nondeterministic. *SFA* separates time complexity of a single NFA to multiple *cDFAs* running simultaneously and cooperatively. In fact, it is also a grouping method, but this grouping is more purposeful and precise compared with grouping in last subsection. The methods in last subsection group with unit of regular expression and different groups have no relationship. While for *SFA*, the grouping works with unit of NFA state, and different groups have connections. However, when encountering a NFA with a large number of conflicting state pairs, the corresponding *SFA* will have massive *cDFAs*, resulting a significant decline in performance.

A recently proposed *TFA* [116] is a real sense of semi-determined FA between NFA and DFA. In traditional automatons, the current matching situation is represented as a DFA state or a set of NFA states which can be as more as the total NFA state numbers. While in *TFA*, the substitution is a set of *TFA* states with upper limitation of b, where each *TFA* state is represented as a combination of selected NFA states. A Set Split Table is generated to map each DFA state (represented as NFA state set) to the corresponding TFA state set. For each input symbol, multiple active TFA states inquires their next NFA states simultaneously, then these NFA states are combined to index the corresponding TFA states as next TFA states. With well-designed encoding, the TFA state number can be orders of magnitude lower than that of DFA while limiting active states to a upper bound of b. In conclusion, *TFA* achieves balance of memory space and memory bandwidth by exploiting sophisticated combinations of NFA states. Compared with other scalable FA solutions, the structure and matching process are more regular, which means that TFA is more suitable to deployed on parallel platforms for acceleration. However, TFA construction involves a time cost NP-hard Set Split Problem, moreover, the algorithm relies on the generation of an integrated DFA.

3) FA Based on Rule Decomposition: As analyzed above, state explosion is mainly caused by metacharacters of wildcard or character class with closures such as "$.*$", "$[a - z]*$" and character class with counting constraints. Some studies [112]–[115], [117]–[119] proposed attaching additional variables or instructions to states to record the matching process instead of using DFA states to represent all the possible matching sequences. We call this kind of FAs as decomposed FAs.

A wildcard closure or character class closure in a pattern can match both the same pattern and the other patterns. Thus, multiple partial matches should be recorded to cover all kinds of input sequences, and each combination of multiple partial match requires a DFA state for representation. When multiple patterns with closures are compiled together, the combination number of multiple partial matches can expand exponentially. Kumar *et al.* [112] proposed a History based Finite Automaton (H-FA) which attaches some states with additional variables to record key events when matching, such as a closure caused by "$.*$". There may be multiple transitions from a state labeled with the same character, while only one transition will be taken based on the variables. At the same time, the variables will be updated during matching process. Take two simple patterns "$ab[\hat{}a]*c$" and "efg" as an example, the corresponding NFA, DFA and H-FA are shown in Figure 18. The DFA has 10 states while the H-FA has only 7 states and a flag. The flag has two states, 0 or 1. The transition from state 0 to state (0, 1) will

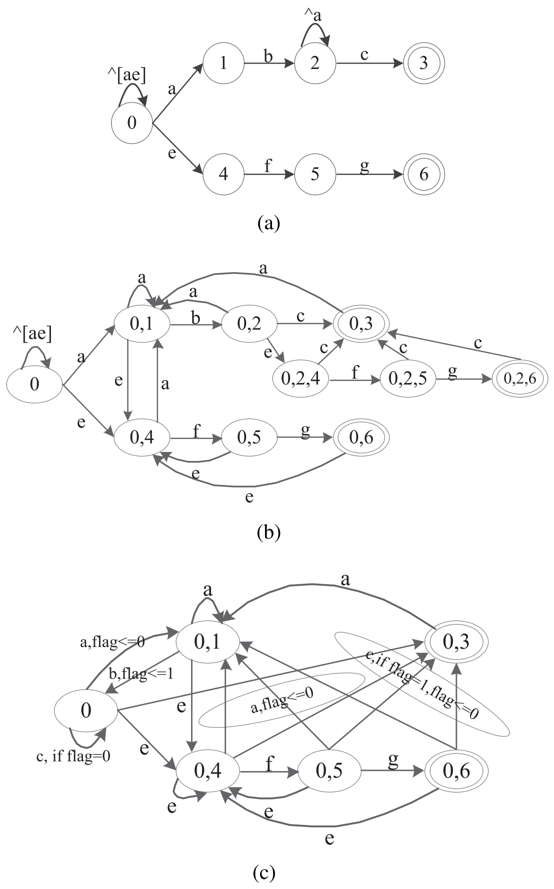


Fig. 18. Corresponding NFA, DFA and H-FA for patterns "*ab*[ˆ*a*]∗*c*" and "*efg*."

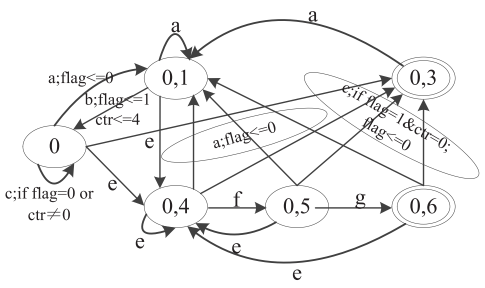


Fig. 19. H-cFA for "*ab*[ˆ*a*]{4}*c*" and "*efg*."

trigger the unset operation, and the transition from state (0, 1) to state 0 will trigger the set operation. In fact, the flag with state 1 means that the matching process meets the sequence of "*ab*" followed by characters of "[ˆ*a*]". When the current state 0 receive the character '*c*', the next transition will relay on the state of the flag, if the flag state is 0, the next transition is state 0, otherwise, the next transition is state (0, 3). The H-FA can accept same sequences as the NFA and DFA, while it has fewer states than DFA and only need access one state for each input character. For character class with counting constraints, DFA also needs large states to simulate all the possible combinations. Kumar introduced counter mechanism to deal with such issues, which generated a H-cFA on the basis of H-FA. Take the patterns "*ab*[ˆ*a*]{4}*c*" and "*efg*" as an example, the corresponding H-cFA is shown in figure 19. In this figure, a counter is responsible for the counting constraints. The transition from state (0, 1) to state 0 set the counter value to 4, and the positive counter is decremented by one for each input character. The corresponding DFA need 20 states for these two patterns, while the H-cFA only needs 6 state, which is even less than NFA states.

Similarly, Smith *et al.* [113], [114] built a formal model *XFA* [113], [114] for this kind of finite automaton. A *XFA* is a 7-tuple (Q, V, Σ, σ, U, (q_0,v_0), F), where V is a set of variables, U is per-state update function Q×V→V which is also called instructions defines how the variables are updated on states, (q_0,v_0) is the initial state and initial variable values, F⊆Q×V is the accepting combinations of state and variable values. An improved feature compared with H-FA is that any state for a given character only has one transition. The construction algorithm first compiles each pattern to individual *XFA*, then combines them to an integrated *XFA*. While the combination operation may cause massive variables and instructions replicating among states, resulting the *XFA* with large variables and instructions per state.

H-FA and *XFA* are the representative solutions for decomposed FAs, however, some inherent weaknesses still exist in practice. First, they require human expert to assist automaton construction, which is inefficient and error-prone. Second, the complex matching process makes it hard for ASIC implementation. Third, the auxiliary variables and instructions will be replicated multiple times when compiling multiple patterns into an integrated FA.

On the basis of H-FA and *XFA*, some novel methods for wildcard or character class closures [115], [117] and counting constraints [118] have been proposed in recent years. Wang and Li [115] also proposed to partition a complex pattern into multiple simple segments and record their connections in a compact table. While this method only works for explosion caused by ".∗". Further, Wang *et al.* [119] extended this method to a complete algorithm PaCC (Partition, Compression, and Combination). First, each regular expression is partitioned into multiple segments without semantic overlapping, thus to guarantee the semantic equivalence. Then, these segments are compiled into individual DFAs with specific compression method. A small and compact relation mapping table (RMT) is employed to record the relationships of these segments which are represented with DFA states. Experimental results show that with the selected Snort and Bro rule sets, PaCC consumes proportional memory space and achieves 1.7 Gbps per core on a HP Z220 SFF workstation.

Recently, Yu *et al.* [117] pointed out limitations of labels for closures in traditional *XFA*. First, for patterns like "*prefix*.∗ *suffix*", there should be no overlapping between *prefix* and *suffix*, as *XFA* matches *prefix* and *suffix* in parallel. Second, for patterns like "*prefix*[ˆ$c_1 \ldots c_k$]*suffix*", if an excluded character c_i occurs in the *suffix*, erroneously match or mismatch may appear for some sequences. For these deficiencies, the author proposed to attach minimum number of labels on generic state transitions to keep functional equivalence. In addition, to avoid massive replications of labeled transitions when merging multiple patterns, the logic associated to labels are moved from

TABLE VII
COMPARISONS AMONG DIFFERENT KINDS OF SCALABLE FAs

Scalable FAs	Construction time	Space requirement	Matching speed	Scalability
Rule Grouping	Very long, as the algorithm needs to compute the interaction relationship for each pair of REs	High or moderate, according to the number of groups and the complexity of single RE	Moderate, due to the multiple DFAs running together	For given grouping number, limited to the scale and complexity of rule sets
Hybrid FAs	Medium or short, according to the deterministic level	Moderate or low, according to the deterministic level	Moderate, due to the NFA or nondeterministic part, but performs well for low matching occurrence scenes like NIDS	Flexible, but large scale and complex pattern sets lead to high nondeterministic level
Decomposed FAs	short, attributed to the complex data structures	Small, attributed to no state explosion	Low, due to massive instruction fetching and variable calculation	Flexible, limited to the complexity of pattern sets

transitions to states by adding some jumping transitions for equivalence. Further, two optimizations named label forwarding and label splitting are proposed to reduce the complexity of the automaton. Experimental evaluation shows that, the proposed automaton provides better memory space consumption and bandwidth requirement than multiple DFA, better worst-case guarantee than hybrid-FA and H-FA, and can also represent the patterns that not supported by traditional *XFA*.

For wildcard or character class with counting constraints, a counter is introduced to restrain state expansion. However, a single counter cannot guarantee full match especially when wildcard or character class overlaps with the prefix. Take a single pattern "$. * a.\{3\}b$" as example, a single counter will miss all the three matches in sequence "*aaaacbbb*". In fact, a new counter should be allocated anytime the matching process matches the prefix of counting constraints to avoid missing any match. Upon processing a character, all activated counter instances need to be updated, resulting massive memory bandwidth consumption. Becchi and Crowley [118] made some improvements by recording the difference of the instance with its previous instance rather than the exact value. As the differences are fixed, only the first counter instance need to be updated for each character, and when the biggest counter complete the counting, the value of next counter is calculated with the difference. In addition, some proposals for counting constraints implemented on reconfigured hardware will be discussed in Section V.

Finally, combined with [119], we make a general comparison among the three kinds of scalable FAs as listed in Table VII. State explosion is the main issue in practice, thus following discussions are made under conditions of large pattern sets. As grouping methods need to generate a DFA for each pattern group, the time cost and memory requirement is very high if the group number is small enough. In addition to DFA generation, grouping process also makes a great contribution to time consumption. Accordingly, efficient DFAs also result in a high matching performance. By comparison, hybrid methods only need to generate part of the DFA, which brings significant savings to time and space consumption. Meanwhile, hybrid FAs can achieve relatively high performance for most benign streams. Decomposed FAs perform best in time and space complexity, but they can not provide satisfactory performance as lots of labels and counters need to be accessed and updated during matching. In addition, automatic construction is more difficult for decomposed FAs compared with grouping and hybrid methods.

V. HARDWARE ACCELERATION

In hardware implementations, the architecture can be divided into two categories, field-programmable gate arrays (FPGA) based architecture and memory based architecture. Existing studies mainly implement NFA or NFA-like FSM on FPGA based architecture while DFA or DFA-like FSM are applied on memory based architecture. In FPGA implementations [120]–[132], each state is mapped to a circuit, and the natural parallelism in hardware circuits can be exploited to overcome the concurrent accesses required by NFA. But FPGA does not support fast dynamic updates, so it is not applicable in network security applications where signature rules are altered frequently. Memory based architecture can be defined as a processing unit with a memory unit, where the processing can be ASICs, Single Instruction Multiple Data (SIMD) processors or GPUs, network processors, and general purpose microprocessors. The memory unit is used to store the SST of FSM, while the processing unit is responsible for packet inspection. Every time an input character arrives, the processor inquires the STT in memory for the next active state(s). DFA or DFA-like FSM is more suitable for this model as it has a deterministic $O(1)$ time cost. And for a given input sequence, the matching speed is only determined by memory access latency. While most high-speed memories have a capacity of not more than 10 Mbyte, which is far less than needed, and they are very expensive in general. Oppositely, latencies for large capacity memories are orders of magnitudes longer, which will severely decrease the matching performance. Current researches exploited massive parallelism in GPU [96], [133]–[141], NPU, General multi-core processors [142]–[148], TCAM [149]–[154] to hide the memory access latency. In practice, thousands to millions of streams are processed at the same time in coarse-grained parallelism. When the current being processed stream is blocked due to long memory access latency, the matching engine switches to the next stream rapidly. The matching speed for a single stream is not improved but the overall throughput is raised to hundreds to thousands times by these means. Some researches

TABLE VIII
IMPLEMENTATION ALTERNATIVES FOR SIGNATURE MATCHING

Parameters		**ASIC**	**FPGA**	**GPU/SIMD**	**Network processors**	**General purpose microprocessors**
Physical constraints	Cost	Highest	Medium	Low	Medium-Low	Low
	Power efficiency	Highest	Low-Medium	High	Medium	Lowest
	Area efficiency	Highest	Worst	High	Medium	Low
	Scalability	High	Low	Medium	High	High
System design	Flexibility	Worst	Medium	Medium	Medium	Best
	Design time	Highest	Medium	Low	Low	Lowest
Performance	Peak performance	Highest	Medium	Medium	Medium	Lowest
	Application performance	Highest	Medium	High-Medium	Medium	Lowest

even designed dedicated ASICs [90], [91], [155]–[160] combined with special FAs and matching mechanisms for a higher improvement.

Smith *et al.* [137] provided a qualitative comparison among these architectures for pattern matching, as shown in Table VIII. Among these platforms, ASICs provide the highest performance and area/engergy efficiency, but they have high system design cost and poor flexibility. FPGAs provide a better flexibility compared with ASIC, but they achieve lower speeds and consumes much more power. On the other hand, general purpose microprocessors has the best flexibility and lowest design cost, but they also achieve the lowest performance. GPUs/SIMDs and network processors perform medially among these architectures. Network processors like Cisco QuantumFlow, IBM PowerNP, Intel IXP target on a superscalar architecture with network-specific functionality to increase efficiency without sacrificing design cost or programmability, and minimize non-recurring cost. Smith *et al.* [137] Both network processors and GPUs/SIMDs utilize massive multi-threading resource to excavate the packet-level parallelism to hide memory access latencies. But compared with the one single instruction on many data streams in GPUs/SIMDs, network processors perform inefficiently as a set of identical instructions are executed on many data streams repeatedly and independently. In the following subsections, we will introduce the merits and disadvantages of the architectures, and analyze their suitability in regular expression matching.

In general, software solutions generate sophisticated FAs and appropriate representations which sacrifice performance for the reduction in storage. While hardwares provide platforms and matching mechanisms for the software solutions, and excavate potential improvement in performance. The links between them are storage organizations and matching mechanisms.

A. FPGA Based REM

NFA based REM requires to maintain a large set of concurrent active states, which brings a big challenge to the memory bandwidth. While, the inherent massive parallel logics in FPGA provide an appropriate platform for NFA to achieve high throughput matching. In 1982, Floyd and Ullman [120] first proposed to translate an NFA to integrated circuits. Sidhu and Prasanna [121] later proposed one-hot encoding scheme to represent NFA states on FPGA, and devised logic structures for basic grammars (as shown in figure 20), which

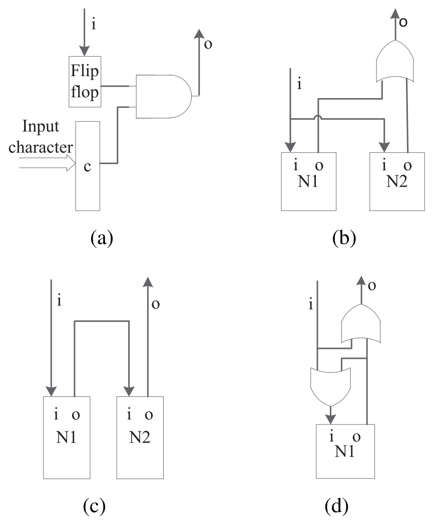


Fig. 20. Logic structure for (a) single character (b) N1|N2 (c) N1N2 (d) N1*.

laid the foundation for NFA based REM on FPGA. Further, Lin *et al.* [122] extended the logic structures implementation for other five metacharacters on the basis of [121]. Yang *et al.* [123] improved the NFA construction algorithm to generate a modular NFA which can be more conveniently translated to FPGA circuits. Yang placed the flip flop after logic gate rather than before it to overlap latencies of character matching and state transition for a higher clock rate.

Despite the inherent high parallelism in FPGA, there are also some inherent challenges in practice. 1) inability of supporting large scale patterns as the limitation of hardware resources, 2) inefficiency in throughput when encountering with large scale and complex patterns, because more patterns occupy more circuits which will cause the decline in chip frequency, 3) time-consuming for updating patterns, because any rule changes will cause recompilation of automaton and the time-cost resynthesis of circuits, 4) unable to support multiple sessions concurrently. As hardware resource utilization and throughput are main issues in FPGA based REM, current researches can be classified into speed-oriented algorithms [124]–[126] and resource-oriented algorithms [127]–[132]. Speed-oriented algorithms mainly relied on multi-stride NFAs to process multi-character per clock, while, multi-stride NFA would definitely cause the inflation of NFA which requires more hardware resources. Resource-oriented algorithms explored more compact hardware representations including more compact structures for NFA, more efficient implementations for complex syntaxes

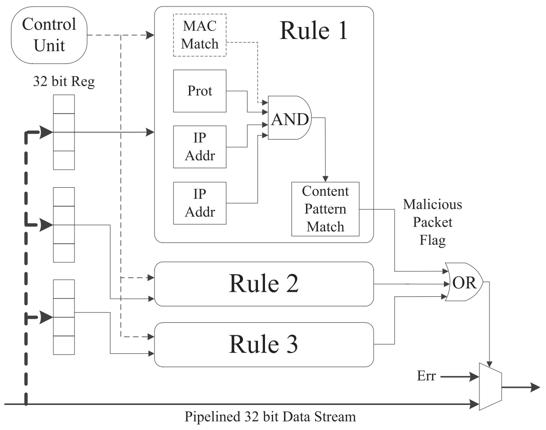


Fig. 21. Parallel processing architecture for Hogwash NIDS.

and shared circuits for different patterns to improve resource utilization. A higher utilization would be benefit for both chip clock and scalability in patterns.

In speed-oriented algorithms, Cho *et al.* [124] first proposed an architecture to process multi-character per clock. Cho *et al.* indicated that traditional simple firewalls can only provide limited security as they only check the packet header, and higher security would depend on multi-layer rule based inspection. It first checks the fields in the packet header, then performs computation-intensive pattern matching process on the payload part. However, most of the existing rule based software firewalls implemented on general purpose CPUs inspect the rules one by one, leading to inevitable performance degradation with the increase in the number of rules. The author devised a parallel inspection mechanism for the Hogwash firewall system [161] based on FPGA, as shown in Figure 21. Hogwash was a lightweight NIDS which employed 105 signature rules to block about 95 percent attacks in 2002. The 32-bit packet stream is dispatched to all rule units concurrently, and each rule unit is responsible for a specific Hogwash rule. It first checks the packet header with the predefined rule header, if matched, the payload data is sent to the content pattern match logic for further pattern matching. In addition to the parallelism of rule processing, four bytes are compared concurrently each time in the content pattern match stage, thus to highly improve the overall throughput. The system achieved a throughput of 2.88 Gbps on the Altera EP20K series FPGA chips. But it only worked for exact string matching, also it did not implement the TCP reassembly function.

Yamagaki *et al.* [125] proposed a novel NFA construction algorithm which iterated the traditional algorithm k times to generate a k-stride NFA. The experiments were targeted on Altera Stratix II (EP2S180) FPGA [162] with 2691 Snort regular expressions containing no quantifiers, and the results show that the 1-stride NFA achieves a throughput of 1.25 Gbps, while the 4-stride reaches 3.63 Gbps at the cost of 20% more LUTs. T.S.H [126] utilized block memory on FPGA to devise a NFA consuming q characters per clock, the results on Xilinx FPGA (Virtex 6) with snort REs also report a similar throughput of 3.2 Gbps, but neither the number of patterns nor pattern characteristics is provided.

Resource-oriented algorithms can be classified into NFA reduction [127], [128], implementations for complex syntaxes [129]–[131] and circuits sharing for subpatterns [122], [129], [132]. Košař and Kořsenek [127] proposed to divide NFA states to conflicting states and no-conflicting states, which are then mapped as NFA and DFA, and the DFA part are stored in block memory. Experiment results on groups of L7 and Snort signatures show that, NFA-Split architecture reduces more than 38% of LUTs and more than 43% of flip-flops for all selected sets of regular expressions only at the cost of a few kilobytes of memory. Košař *et al.* [128] improved NFA reduction algorithms before mapping it to FPGA, and results show that it was able to decrease the number of LUT-FF pairs by 65.55% for Snort ftp signatures.

Despite of NFA reduction, many studies emphasize on optimizing implementations for complex syntaxes especially for character class with counting constraints. Traditional methods for repetitions were duplicating the sub-expression circuits multiple times which was too luxury to implement on FPGA. Bisop *et al.* [129] exploited Xilinx shift register lookup tables (SRL16) to implement single character repetitions. Experimental results show that this method can support 500 REs from Snort v2.4 using 25K logic cells, and achieves 2 Gbps throughput on Virtex2 device and 2.9 Gbps on Virtex4 device. Further, Faezipour and Nourani [130] introduced a new basic building block to handle repetitions for arbitrary sub-expressions rather than just a single character, while it might have uncertainty when encountering with overlaps between adjacent character classes. Simulation results verify that this approach achieves 40% area savings for the entire Snort v2.7 set without sacrificing the performance. Wang *et al.* [131] proposed a counter based algorithm named MIN-MAX to handle repetitions of complex character classes, where the massive circuits were replaced by counters in block memory. It is proved that MIN-MAX also supports overlapped matching when REs are inherently collision free or safe. A case study on subset of Snort REs shows that, 74% memory bits can be saved compared with conventional NFA-based designs.

Sharing sub-expression circuit is another way to save circuits, Bispo proposed prefix sharing in [129], while infix sharing and suffix sharing is hard to implement because the sharing circuits must remember the before matching paths. Lin *et al.* [122] and Hieu *et al.* [132] exploited implementations for infix and suffix sharing. Lin *et al.* [122] argued that his proposal can achieve an average of 28% in area reduction on Snort rule sets and 38% on the patterns from the industry company Trend Micro. In addition, the circuit delay is also improved as it reduces the fan-out load of the payload input and routing complexity, the approach contributes to an average of 22% in circuit delay reduction both on Snort rule sets and Trend Micro sets. Hieu *et al.* [132] proposed an efficient NFA-based regular expression matching engine (ENREM) on FPGA, the ENREM employs a novel infix and suffix sharing architecture with several optimizations to reduce the required circuit area. The experimental results with Snort rule sets on Xilinx Virtex-II Pro XC2VP-50 FPGA show that, compared with traditional approaches ENREM can reduce 42%

TABLE IX
CURRENT RESEARCH PROGRESS ON FPGA PLATFORMS

Algorithm	Main issue	Implementation Device	Pattern characteristics	Resource saving ratio	Reported throughput
ENREM [132]	infix and suffix sharing	Xilinx Virtex-II Pro XC2VP-50 FPGA	8 original rule sets and 3 extracted rule sets in Snort v2.9	42% LUTs and 32% FlipFlops	1.45 to 2.35 Gbps
NFA reduction [128]	state and transition reduction	Xilinx Virtex-5 155 LXT FPGA	25 randomly selected modules of the Snort IDS (not clear)	66% for a Snort ftp module	Not mentioned
MIN-MAX [131]	compact representation for character class with counting constrains	Virtex 5 evaluation board (XUPV5-LX110T)	1667 regex rules in S-nort (May 2011)	74% memory bits compared to conventional NFA-based designs	Not mentioned

LUTs and 32% FlipFlops while maintains throughput from 1.45 Gbps to 2.35 Gbps.

For all the related studies, the credible throughput for a medium rule set based on FPGA circuits was not more than 10 Gbps. Table IX presents current research progress on FPGA platforms in recent several years.

B. GPU Based REM

With the rapid deployment of GPU in various scientific applications, some proposals [96], [133]–[141] targeted GPU as REM platform for its massive hardware parallelism (more than 1000 threads) and high memory bandwidth (almost 100 Gbps). The nature is to utilize the massive threads to hide latencies caused by global memory accesses. Firstly, we will introduce the architecture of GPUs, mainly NVIDIA GPUs with the programming model of Compute Unified Device Architecture (CUDA). Then, the general REM process on GPU and the main issues are discussed. Finally, we will evaluate the related works, and provide guidelines for building good REM engines on GPU platforms.

1) Introduction to GPU Architecture: GPU is comprised of a set of streaming multiprocessors (SMs) where each SM is a multiprocessor with multiple processores running in single instruction multiple data (SIMD) mode. A unit of work issued from the host CPU to GPU is called a *kernel*, and the kernel is executed as many different threads organized in *thread blocks*. Each multiprocessor can execute multiple thread blocks, and they are further organized into *wraps*. Each *wrap* is a potion of active group that contains same number of threads, and the threads from a same *wrap* are executed by one multiprocessor in a SIMD mode. Further, multiple active wraps are scheduled in time-sliced mode, thus to fully utilize the computational resources of GPU.

The processors within a same SM share the instruction unit, thus they can execute the same instruction simultaneously just through assigning different execution context to different threads. But when any control flow instruction leads the threads to different execution paths (eg, due to the different outcomes of a conditional branch), a *wrap divergence* occurs. The wrap divergence can highly reduce the overall throughput, because the threads must be serialized, and the threads goes back to the same execution path only when all the threads reach to the same end [133].

The layered memory of GPU is consisted of low latency on-chip *shared memory* (each for a SM), high latency off-chip global memory, constant memory, and texture memory. The shared memory can be manipulated explicitly by the programmer, and threads in the same block communicate with the shared memory and threads in different blocks have no interaction. The *constant*, *texture* and *global* memory can be read or write by the host computer, but the content is persistent across kernel launched by the same application [134]. In addition, as texture and constant memory are cached, reads from them are much faster than the global memory.

2) Matching Processes: In REM, GPU is the coprocessor of CPU. CPU is responsible for pattern compilation, pattern deployment, packet receiving and transferring. CPU sends the compiled FSM and packets to GPU, GPU focuses on pattern matching and transfers the matching results back to CPU. Refer to [133], we devised a figure, as shown in Figure 22, to provide a classical matching process on GPU platforms, Next, we will discuss these processes in detail.

The first step is to collect packets from network interface and transfer them to GPU memory. Considering the overhead of transferring each packet separately, small transfers are often merged into a larger one and transferred to the GPU in a batch. In addition, packets need to be organized into groups according to their five-tuples (source IP, destination IP, source port, destination port, protocol number), because applications like snort organize the patterns in groups and each packet only need to matched with the patterns that have same five-tuples. The transfer becomes complex when the signature in a TCP stream cross through multiple packets, then the TCP stream reassembly function is required to reconstruct tcp stream as in modern NIDS. In this situation, a thread needs record the partial matching results and keep on processing the subsequent packets from the same stream sequentially.

The second step is pattern matching on GPU, and this involves the deployment of automatons and packets, and the configuration of parallel matching process. As has declared above, wrap divergence would highly decrease the overall performance, it is crucial to control the divergence as less as impossible. The AC algorithm in string matching and DFA in regular matching are suit for this architecture, because each time they read a byte and inquire the transition table to move to the next state, there is no divergence during the matching process. The deployment of DFA state transition table and

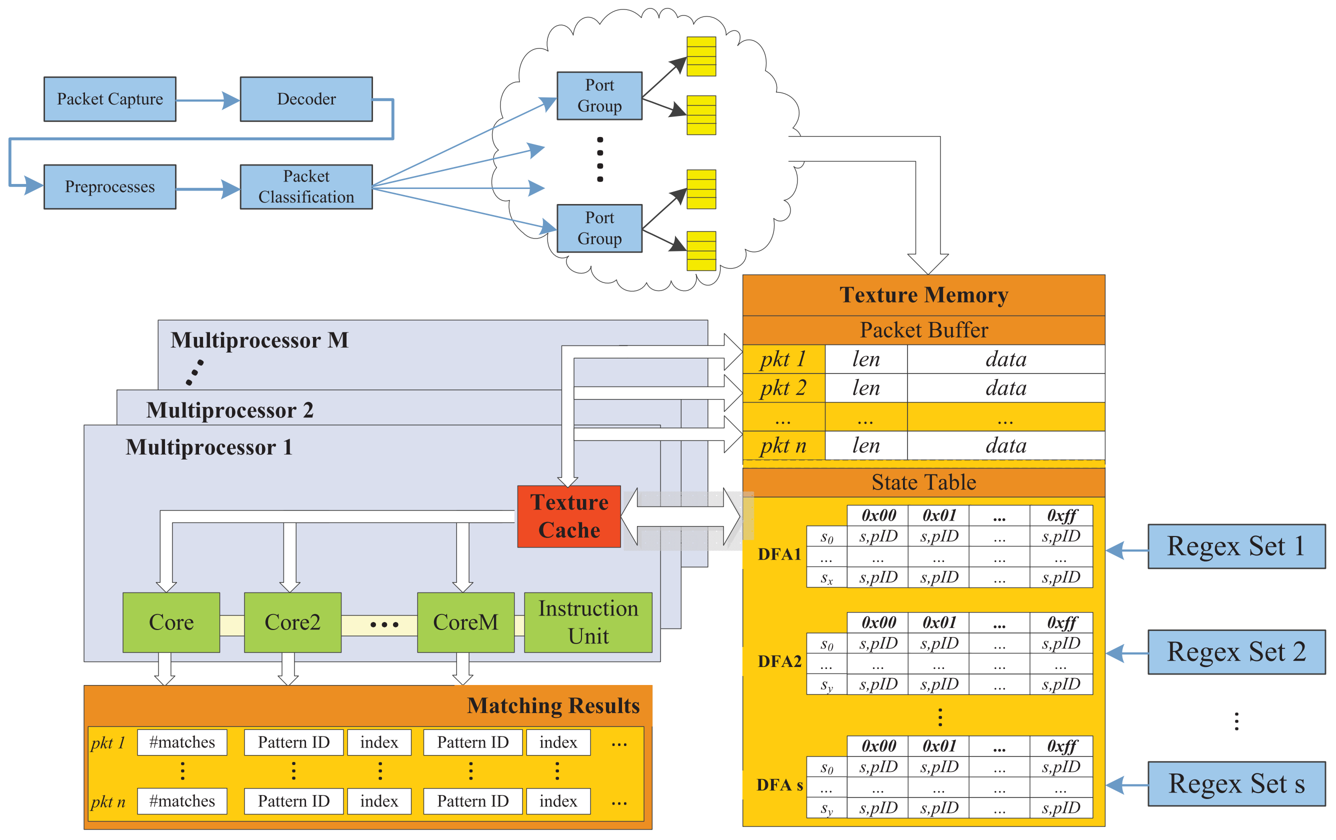


Fig. 22. A classical implementation of regular expression matching for DPI on GPU platform.

packets is another issue needs attention, and texture memory performs better than global memory for two main reasons. First, texture memory can be read in a random mode, which is suit for DFA matching as its irregular access across the transition table, while the accesses in global memory must be coalesced [163]. Second, texture memory is cached, which largely reduce the latencies when cache hit occurs. But when state explosion occurs, the global memory also should be selected for automaton storage.

Accesses to packets are regular and sequential, so multiple bytes can be fetched once to avoid latencies and bandwidth consumption. While, accesses to automaton are irregular and random, because state migration is erratic. To fully utilize the high memory bandwidth, hardware controller tries to coalesce smaller accesses from different threads into fewer and larger accesses [135]. The limitation is that, the coalesced accesses must lay in the same naturally-aligned 256-byte area, thus how to organize the automaton and matching algorithm to meet the limitation is another main design issue.

Among current researches, many parallel mechanisms have been proposed on GPU platforms, which will be discussed in the next subsection detailedly. A classical method is to assign a single packet to each thread. Each thread block contains same number of threads and processes equal number of packets in a batch target. A minor disadvantage is that, the packet length may vary in the same thread block, thus all threads must wait until all threads have finished. The final step is to transfer the matching results back to the host CPU. Matching results for each packet will be written in specified positions and copied to host memory when the kernel finishes.

3) Researches: Gnort [133] proposed by Vasiliadis is the first prototype of GPU based intrusion detection system. The multi-pattern AC matching algorithm [1] is implemented on NVIDIA GeForce 8600GT card (32 stream processors organized in 4 multiprocessors, operating at 1.2 GHz with 512 MB of memory) and Intel Pentium 4 processor (3.4 GHz, 2 GB memory), and the system reaches a maximum throughput of 2.3 Gbps with 1000 Snort string patterns whose size varied from 5 to 25 bytes. While AC is only useful for exact string matching, the complex regular patterns are left to CPU for processing. Furthermore, the author also realized that current motherboards supported multiple GPUs, thus it was feasible to build the clusters of GPUs to achieve faster intrusion detection systems.

Latter, Vasiliadis *et al.* [134] extended the architecture of Gnort for supporting both string and regular expression matching. The author indicated that exact strings could not fulfill the description of signatures completely in current NIDS for two reasons. First, strings are unable to accurately describe the attack characteristics, this may increase the number of false positives. Second, the same string literal maybe shared among multiple rules, which would also bring false positives. On the contrary, regular expressions are much more flexible and expressive than simple strings, they can support a much wider attack signatures, thus they are widely used in NIDS, spam filtering, and virus scanners applications.

Each rule in Snort NIDS is composed of two parts, the rule header and rule options. The rule header describes the packet header informations such as source and destination IP addresses, ports, transport layer protocols, as well as the actions when matched, while the rule options mainly describe the payload signatures. There are two main fields in rule options, content field for string matching and pcre field for regular expression matching, and early versions of Snort only support string matching. Each rule is compiled separately in Snort NIDS, thus it had to match a given packet against all the regular expressions that fulfill the packet header one by one. To reduce the number of regular expressions to be matched, Snort employs string matching to pre-filter the regular expressions that can not be matched. Take the following rule for example, "alert tcp any any -> any 21 (content:"PASS"; pcre:"/^PASS\s*\n/smi";)". The 'PASS' in content field is an exact sub-string of the pcre field, thus the packets match the regular expression of '/^PASS\s*\n/smi' must contain the 'PASS' string. In other words, if the content field is not matched, the pcre part could be never matched. Snort extracts fixed string in each regular expression and compiled them into an integrated FSA, then only those rules whose content field are matched require further regular expression matching.

Vasiliadis *et al.* [134] adopted the same matching mechanism, string pre-filter is conducted in CPU, and GPU is responsible for regular expression matching. But not all regular expressions are processed on GPU, it compiles each regular expression to a separate DFA with the upper limitation of 5000 states (remaining expressions are processed by CPU with NFAs). Regardless of the packet transfer overhead, the system reaches at most 16 Gbps with an NVIDIA GeForce 9800 GX2 card (two PCBs, each is an Geforce 8800 GTS 512) and regular expressions of Snort 2.6 deployed in texture memory. The author also evaluated the overall performance of the Snort IDS, although the overall performance only reaches 800 Mbit/s, it is still 8 times higher than the CPU implementation. Further, even matching each packet with 20 regular expressions, the overall throughput can remains over 700 Mbit/s on GPU.

Vasiliadis *et al.* [136] also explored the performance effect of different memory hierarchies on a new Fermi architecture GPU, named GTX 480, and some valuable experiments were concluded on this device as follows. First, when fetching packets from GPU global memory, 16B per time per thread results in a significant increase for processing performance, as it greatly reduces the number of memory transactions. Second, when the thread number is moved from 12288 to 16834, a throughput degradation is observed, thus the thread number of 12288 is a good configuration for GTX 480. Third, both texture and global memory are equipped with caches in Fermi architecture, thus utilizing both of them can reach a better performance. In fact, the experimental comparisons show that global memory fit better for state table accesses, while packet accesses performs better on texture memory. Finally, the scheme achieve a throughput of 6.49 Gbps for small packets and 29.7 Gbps for full-payload packets with pure-DFA solution.

Smith *et al.* [137] analyzed the characters of GPU from memory, control-flow and concurrency, demonstrating that GPU is suit for REM. DFA is most suit for GPU acceleration as it has no divergence in searching phase, but DFA has the state explosion problem. Smith *et al.* [137] implemented a matching prototype system on G80 GPU with both DFAs and *XFAs* representation for three signature sets of FTP, SMTP and HTTP from both Cisco Systems [164] and Snort, and compares with multiprocessors of Pentium4 and Niagara. Results show that, GPU solutions can achieve an average 8.6× and 6.7× speedups for DFAs and XFAs. However, Yu and Becchi [138] later indicated that XFA is only suit for regular expressions that can be broken into non-overlapping sub-patterns.

Cascarano *et al.* [135] proposed a novel NFA based matching engine named iNFAnt on GPU. As NFA has no state explosion, large and complex rule sets such as Snort and L7 can stored in small texture memory. In addition, massive hardware parallelism can be employed to track the active NFA states to offset the traversal cost for each input symbol. To efficiently coalesce memory accesses, the author designed a sophisticated NFA representation called symbol-first representation where transitions were stored as <source, destination> and sorted by the triggering symbol. In the matching algorithm, each packet is assigned to a different thread block for coarse-grained parallelism. For a given symbol, each thread in the related block is responsible for a transition of this symbol for fine-grained parallelism. As transitions for the same symbol are adjacent in memory, the accesses can be easily coalesced. The traversal algorithm omits much divergence in traditional NFA matching process, and exploits all the available bandwidth efficiently. The experiment compares iNFAnt on an nVidia GeForce 260 GTX graphics card with HFA [95] on a 4-core Xeon machine running at 3 GHz, with a simple http set, a moderate Snort 534 set [165] and the complex L7-filter set. The throughput of non-stride NFAs are comparable to though lower than HFAs, but with the assistance of multistriding and self-loop optimizations, they performs much better than HFA solutions. However, the overall throughput only reaches about 1.5 Gbps for Snort534 and 1.0 Gbps for L7-filter at most.

Zu *et al.* [139] declared that they solved the limitations of iNFAnt, namely the poor worst-case behavior and unpredictable performance. The main idea is divide the overall NFA states into compatible groups, where states in the same group can not be active concurrently. This idea is similar with the SFA [96] proposed by Yang. But as we have analyzed, it is a huge task as the classification involves the exploration of all possible NFA activations, namely the DFA generation process. Thus, it is only suit to small and simple rule sets without state explosion. Therefore, the comparisons with iNFAnt is not unfair only on small and simple rule sets.

Lin *et al.* [140] proposed a novel parallel exact string matching algorithm named Parallel Failureless Aho–Corasick (PFAC) on GPUs. To efficiently exploit the massive parallel resources, the author introduced a series of optimizations, including reducing the latency of transition table lookup, reducing global memory transactions, eliminating output table accesses, coalesced writing to the global memory, avoiding bank conflict of shared memory, and enhancing communication between CPU and GPU. The experiment was conducted on NIVIDIA GTX580 GPU with string patterns from Snort V2.8 and input traces from DEFCON [166], and the performance reaches a throughput of 143.16 Gbps, 14.74 times higher than the baseline system with original AC algorithm on 4 cores of Intel Core i7-950. However, this algorithm only works for exact strings matching.

Yu and Becchi [138], [141] indicated that most current GPU based solutions aim at achieving good performance on small rule sets, and iNFAnt is the first proposal that can be easily applied to pattern sets of arbitrary size and complexity. The author focused on the regular sets with practical complexity and size, and explored the advantages and shortages of different automaton implementations based on GPU. By observing that only several NFA states are active in most processing steps and NFA states can be divided into compatible groups, Yu applied three optimizations to the original iNFAnt, as detailed in [138]. Further, the matching efficiency of grouping DFAs (uncompressed) [167] and the compressed A-DFA [106] are analyzed detailedly from algorithmic perspective, and an enhanced compressed DFA is proposed to reduce warp divergence and thread underutilization of compressed DFA. Final experiments were conducted on CPU (Intel Xeon E5620) based NFA and DFA, and GPU (NVIDIA GTX 480 GPU) based NFA, optimized NFA, DFA, compressed DFA, enhanced compressed DFA, with different level of pattern sets and trace sets. Results show that the original DFA outperforms other solutions for its computation regularity, and it is scalable to the number of packet flows that processed in parallel. However, it is not scalable to complex and large rule sets, as the space requirement may far exceeds the capacity of GPU memory. In addition, throughput of these solutions are all below 0.3 Gbps for the tested rule sets.

In a nutshell, GPU is an available device for REM as high latencies can be hidden by massive threads. The limitations for GPU are divergences in searching phase and memory access coalescence, both of which are related to automaton representation. DFA and NFA have opposite characteristics in automaton size and matching divergence. The authentic performance [138] for large complex rule set based on GPU is not more than 0.3 Gbps.

C. General Purpose Multi-Core Processors Based REM

Multi-core processor is a basic and widely used parallel platform. Though the number of parallel execution units is orders lower than GPU, multi-core processor has much more hardware resources for each unit, such as caches, registers, higher clock frequency, and more flexible execution model. Thus, it is convenient to employ the multiple cores as parallel matching engines for pattern matching.

A straightforward method to exploit parallelism is to assign one flow to each execution unit, and multiple flows are processed in parallel and independently. Scarpazza *et al.* [142] implemented a DFA based pattern matching system on IBM Cell Broadband Engine processor, which contains 8 processing elements called SPE. Each SPE is comprised of a SIMD processor, some registers, local store memory (256KB), etc. As the main memory cannot be accessed directly by SPE, only the local store memory can be used for STT storage. Each SPE is assigned a distinct flow for matching, parallelism exists between different SPEs for different flows. In addition, the author further exploited the SIMD characteristic to process 16 flows in parallel on one SPE, which achieved another 2.51 times performance improvement than sequential processing. As local memory is very limited, the throughput can reach 40 Gbps for small DFAs with not more than 1500 states. For larger DFA sizes, it needs assistance of dynamic STT replacement from main memory, which will cause serious performance degradation.

Further, Jiang *et al.* [143] proposed a similar parallel architecture called Parallel DFA (PDFA). As in traditional parallel matching algorithm, the DFA is stored in an identical memory module, which can be accessed only once every time. When multiple access requirements from different engines arrive at the same time, these accesses must be processed in a sequential manner. Jiang devised a mechanism to split the DFA into multiple partitions and mapped them into different memory modules which can be accessed concurrently. Thus, multiple accesses can be distributed to different modules to extract the potential parallelism in memory access. The experiments for Snorts web rule set and L7-filter reaches a throughput of only about 1 Gbps.

The above articles [142], [143] exploited parallelism among different flows, modern researches [144]–[147] also focused on parallelism inside the same flow. In traditional methods, the characters in a flow must be processed one by one, because the current state can be only achieved from the previous state and character, thus the time complexity is definite $O(n)$ for input size of n. Holub and Štekr [144] firstly proposed to divide a flow into multiple chunks without overlapping, each core is assigned a chunk and the chunks are processed in parallel. Except for the first chunk, no core knows the initial state for its chunk. Thus, each chunk must assume all the DFA states as initial states and makes mappings to all the corresponding final states which will bring huge computation cost. Then, the partial results are joined to achieve the final state. The speedup is $O(|P|/|Q|)$ where $|P|$ is number of cores and $|Q|$ is the number of DFA states. This means it is effective only when DFA state number is less than the core number. To break the limitations of large assumed initial states, the author exploited the k-local characters of the DFA. K-local automaton means that, any state will transfer to the same state for an identical string with length of k, and all k-length strings have this feature. If a chunk is assigned additional k last characters from the previous chunk, it is easy to get the definite initial state without assumption. Though the parallel run of k-local DFA can reach a speedup of $O(|P|)$, it is apparently rare to encounter such kind of DFAs in practice.

Instead of employing all states as initial states for a chunk, Luchaup *et al.* [145], [146] proposed a speculative parallel pattern matching (SPPM) approach which only speculatively selects a definite state as initial state for all the chunks. The observation is that a few hot states are destination states for most transitions, the access frequencies are orders of magnitude than other states. In addition, each processing unit should keep a history buffer to save all the traversal states corresponding to each character of its chunk. The validation process is sequential from first chunk to last chunk. For a given chunk, the validated last state of its previous chunk is compared with the assumed initial state. If equal, the speculation is right and the last state in its history buffer is a validated state. Else, the chunk should be rematched from the right state until the current state for a character is equal to the state in the same position of the history buffer. If this happens, the rest characters need not to be rematched and the last state is still a validated state. The worst case is that a whole chunk is rematched, but this rarely happens. Performance gains from quick validation rather than accurate speculation for initial states. Experiment results show that average rematched length for a chunk is about 2, which is far less than chunk size and will result in a near linear speedup.

Ko *et al.* [147] proposed another approach called reverse lookahead symbols to limit the number of speculative initial states. Rather than assuming all states as initial states for a chunk, the author utilize the last few characters from previous chunk to eliminate some states which can never be the initial states for this chunk. As given the character sequence, none of these eliminated states can be reached from any state, while this elimination may introduce massive computation. In addition, the author also discussed methods for balance workload among processing units.

Further Najam *et al.* [168], combined the speculative idea with multi-stride DFA, aiming to achieve a higher speedup. While, the multi-stride representation will definitely result in large expansion, even the authors declared that they propose a transition compression algorithm using alphabet compression table to limit the memory usage of multi-stride DFA. Thus, this proposal only works for very small DFAs, as we speculated, the biggest DFA in experiments only contains 13490 states.

Articles in [142]–[147] all employed DFA as automaton representation which cannot handle the state explosion. Yang and Prasanna [148] exploited a NFA-like representation *SR*-NFA on multi-core structure. With assistance of rule grouping, the compact *SR*-NFA can be entirely stored in the level-2 even level-1 cache, which will contribute much to performance improvement. The prototype on an 8-core 2.6 GHz Opteron platform can achieve 2.2 Gbps throughput for large rule sets.

The above researches in this subsection only focus on exploiting parallelism for the pattern matching process in DPI applications, and most of them only employ string or regular expression signatures of Snort to detect whether a packet contains attack characteristics. Simple packet based pattern matching can only achieve limited and stateless detection, modern network security monitoring systems require sophisticated analysis of protocols at a higher semantic level, even incorporating context correlated across multiple connections and hosts [169]. Sommer listed a series of applications that require global analysis, for example, worm scan detection [170] requires to record the initiation requests of all connections; contact graph analysis [171] is used to detect new worms, but it requires a global connection information during a time window; stepping-stone detection [172] requires correlating packet timing across multiple connections.

Signature matching is only a small portion of the whole process, for example, the preceding TCP stream reassembly is even more difficult than simple string matching. Thus, parallelism exploitation based on multi-core platforms should be conducted from a global perspective, not just for the pattern matching part. To fully exploit the potential global parallelism, Paxson *et al.* [173] divided the process of high level network analysis applications into multiple stages and discussed them in detail, as shown in Figure 23. The rectangular boxes represent different stages of process, and the semantic level of them is increasing from left to right. The arrows indicate the data flow from low level to high level, and the thickness of an arrow represents the magnitude of data flow or how many threads are needed to process them concurrently. Furthermore, fan-out arrows represent that for a given data flow multiple analysis can be executed concurrently on multiple threads, and fan-in arrows represent that multiple data flow should be gathered for higher level analysis. And the numbers below each stage indicate the magnitude of relevant parallelism that can be achieved under a 1-10 Gbps link.

To exploit the potential parallelism for each stage, we need to discuss them in detail. The first stage called stream demux is a sequential process, which demultiplexes the received packets for per-flow analysis. For a link of 1-10 Gbps, the magnitude of flows would be about 10^4. This means that the next TCP reassembly stage for these 10^4 independent streams could be performed concurrently. Even commodity multi-core platforms do not contain so many threads, each thread can still process a subset of these streams independently and parallelly. Then the next stage is the high level protocol analysis, today the simple port number is inadequate to identify most of the application protocols. A better method is to run multiple possible protocol parsers concurrently to confirm which application the flow is. Thus the protocol analysis stage can provide much more parallelism than TCP stream reassembly, in this figure it is labeled as 10^5. After the protocol analysis procedure, only one application protocol is selected as the protocol of the given flow, thus the fan-in arrows indicate that the parallelism of next stage declines to the same as TCP reassembly.

Through the application protocol analysis, we can extract a series of application-level activities such as the parameterization of requests, error conditions, status codes associated with replies, signature matches, items, and so on [169], which are also called ‘events’. Then for the next stage, multiple events which come from different flows but share the same host or flow type could be gathered for an aggregate level analysis. Sommer explained this with the example of scan detection, it is necessary to gather statistics for a given host, how many servers it tried to connect and with what methods. Finally, at a even higher level of global analysis, we gather the events not only from the same host but also from multiple different hosts. ‘Content sifting’ [174] is a classical example for this level, in order to detect the propagation of the worms, it needs to analyze multiple traffic flows from different hosts.

In a word, to fully utilize the power of multi-core platforms for DPI applications, we must structure the applications in parallel from a global perspective, divide the procedures

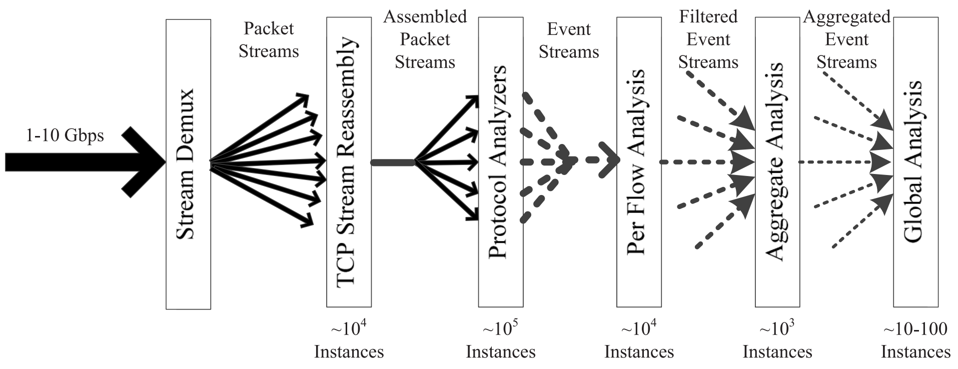


Fig. 23. The parallelism exhibit in a high level network analysis pipeline.

into multiple stages where each stage can be executed concurrently. Furthermore, we should also pay attention to the memory access patterns of the applications, not only to the execution models, as the memory access speed can completely dominate the final performance. Here we do not extend this issue for space limitation, interested readers may refer to [169] for deeper understanding.

In a nutshell, general purpose multi-core processors does not work as well as GPU devices. Compared with GPU devices, they have more advanced hardware characteristics, like larger caches, higher clock frequencies, and more flexible execution models, but the parallel resources are orders lower. The advanced features of multi-core processors only benefit to the speedup of a single thread, not to the overall throughput. In addition, the high-speed large cache memory of multi-core processors can not be managed by user, thus it is not convenient to utilize hot states to achieve a high cache hit ratio. Current researches mainly focus on speculative matching, which also only benefit to the single stream not the overall performance. In our opinion, general purpose multi-core processor is not a recommended platform compared with the more powerful GPU devices.

D. TCAM Based REM

Ternary content addressable memory (TCAM) is widely deployed in network devices for packet classification and routing. TCAM is kind of memory consisting of a set of entries, and all the entries can be searched parallelly with high speed. Each entry is a bit vector, where each bit has three states, namely 0, or 1, or '*' (do not care). In addition, the entries are usually organized according to their values, namely the upper entry with the smaller value. As the existence of '*' bit, an input value may match multiple entries. TCAM always returns the index of the first matched entry, namely the smallest entry index. A basic TCAM matching example [149] is shown in Figure 24, for input value '1000', TCAM return the first matched entry '100*', namely the 2^{th} entry. In addition, the entry width is configurable, according to the customer requirement.

In today's network environment, thousands of new malicious worms and viruses spread fast everyday. Traditional prevention method is an end-host based mechanism, which

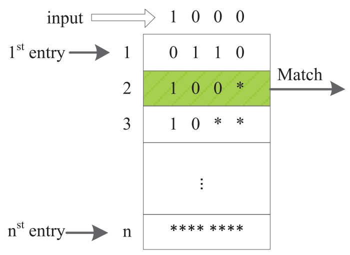


Fig. 24. A simple example for TCAM lookup.

installs new patches or updates the security software for individual hosts. However, this method is inefficient when encountering with new fast-spread worms, as the spread speed is far more faster than the reaction speed of end hosts. It is too hard to install new patches or update the security database for an enterprise with large clients on a short time. Thus the network based detection mechanism is an effective supplementary for end host detection, and it relies on inspecting packet payloads at critical forwarding points, mainly the routers. However, pattern matching is a rather time-cost procedure, traditional software-only method can not fulfill the ever-increasing link speed. For the wide deployment of TCAM in current network devices, it has also been targeted to accelerate the pattern matching process.

When replaced with the string patterns, the TCAM can be utilized for string matching. Gokhale *et al.* [175] employed binary CAM for string matching, for m string patterns with fixed length of w, the method occupies mw bytes of CAM space, and provides a deterministic time complexity of $O(n)$ for any input sequence with length of n. Later, Yu *et al.* [149] proposed algorithms for arbitrary long string patterns, even for complex patterns like negation and correlated patterns. Based on the 1768 string patterns of ClamAV [176] virus database, the author devised a packet based anti virus system which achieved nearly the OC-48 line rates (2.5 Gbps) with only 240 KB TCAM space.

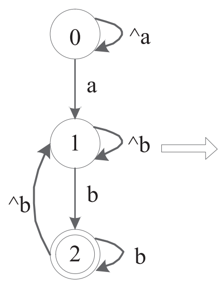
Traditional algorithms lookup the TCAM one time for each input character, Sung *et al.* [150] pushed a step further to process multiple characters for each TCAM lookup. The algorithm uses an m-byte jumping window pattern-matching scheme, at the cost of enlarged TCAM space requirement.

This scheme reported a throughput of 10 Gbps for 2247 string patterns from Snort, with the TCAM space of 9 Mbit. Alicherry *et al.* [151] also devised a novel multiple string matching algorithm which can process multiple characters each time, and some optimizations for memory requirement reduction were also provided in this paper. Experimental results show that, this mechanism works well for virus datasets of ClamAV. Yun [177] devised a novel state encoding scheme for TCAM based Aho–Corasick multistring matching algorithm. On the other hand, Zheng *et al.* [178] focused on how to develop parallelism as well as improve power efficiency for TCAM based string matching. Based on the work of [149], Zheng proposed to partition both individual flows and the pattern set. Flow partition indicates dividing a flow into multiple sequential segments for parallel matching, while pattern partition means group the pattern set into multiple subsets thus they can be matched individually and selectively. These early TCAM based pattern matching methods [149]–[151], [175] are devised mainly for exact string matching, even with some extensions [149], these methods cannot support all the metacharacters in regular expressions.

Unlike string matching on TCAM, regular expression matching on TCAM mainly employs the DFA representation. Meiners *et al.* [154] summarized three reasons for why TCAM is suitable for regular expressions matching. First, the ternary nature and first-match semantics make it capable for encoding a large DFA. Second, except for the low memory access latencies, any memory access can finish in a constant time with its high-parallel lookup ability, regardless of the entry number. Third, TCAM is an off-the-shelf chip that has been widely used in modern network devices.

In DFA based regular expression matching mechanism on TCAM, the transitions are organized as entries for matching. Each entry is physically consist of a TCAM part and an associated SRAM part, where the source state and input symbol are stored in TCAM for index, and the destination state is stored in SRAM. Take a simple pattern "$a.*b$" as an example, the corresponding DFA and TCAM table are shown in Figure 25. The ternary bit '*' can match both '0' and '1', which is very beneficial to DFA compression. As most DFA states have only a few destinations, even with no other compression techniques, TCAM encoding can achieve a big compression ratio without sacrificing performance. In this example, each state can be represented with two entries. In matching process, the current state and input symbol are compared with all the TCAM entries concurrently, and the position of the first matched entry is returned in one cycle. Then, the position is used to access the corresponding SRAM part for the next state.

Though the basic encoding can achieve much compression, it is still a great challenge to encode large DFA in small TCAMs, even given the largest available TCAM chip with the capacity of 72 Mbit, not to mention that the majority of its space will be used for routing and flow table management. Efforts have been made to further merge entries [153], [154] and avoid state explosion [152], [154]. Entry merging utilizes the ternary characteristics and longest prefix match semantics of TCAM to merge transitions of intra-state or inter-state.



TCAM		SRAM
Src ID	Input	Dest ID
00	0110 0001	01
00	**** ****	00
01	0110 0010	10
01	**** ****	01
10	0110 0010	10
10	**** ****	01

Fig. 25. DFA and TCAM organization for pattern "$a.*b$."

TCAM		SRAM
Src ID	Input	Dest ID
00	0110 0001	01
00	**** ****	00
**	0110 0010	10
**	**** ****	01

Fig. 26. The improved TCAM organization with shadow encoding for pattern "$a.*b$."

Thus, traditional compression algorithms such as D^2FA [100] and σFA [103] are not suitable for TCAM encoding.

Meiners *et al.* [153] proposed two approaches to reduce the memory requirement of DFA representation on TCAM, separately named as *transition sharing* and *table consolidation*. The basic idea is to merge multiple DFA transitions into one entry by leveraging the ternary characteristics of TCAM and redundancies in DFA transition table. Meiners *et al.* [153] divided the redundancies into two categories, the character redundancy where transitions share the same source state and destination state while differs in the labeled characters, and the state redundancy where transitions share same labeled character and destination state while differs in source states. Transition sharing method includes *character bundling* method and *shadow encoding* method. Basic encoding (Meiners *et al.* [153] named it as *character bundling*) can only merging intra-state transitions, namely the character redundancy, through introducing ternary bits in input symbol field. In practice, massive redundancy still exists among states, for instance, state 1 has the same destination state for labeled transition 'a' and default transition with state 2. Meiners *et al.* [154] proposed *shadow encoding* algorithm to reduce intra-state redundancy by introducing ternary bit in source state filed. With shadow encoding, the entries in Figure 25 can be further organized as in Figure 26, where two more entries can be saved. The sophisticated shadow encoding algorithm involves determining the best order of state tables organized in TCAM, identifying the entries to remove, choosing binary IDs for each state. Readers could refer to the more complex example in [153] for a better understanding.

Table consolidation means merging transition tables of multiple states into a single one. The motivation behind this idea is that different transition tables may share similar structures like the common entries, even they do not have the same decisions. Due to these shared structures, multiple similar entries can be merged into one entry, and the decision part in

SRAM maintains multiple destination states for these merged transitions, thus to reduce the space requirement of TCAM entries. Table consolidation is also about merging entries, but it is a state-grained merging, while the transition sharing is only a transition-grained merging. Meiners also indicated that the origin of similar structures comes from the '.*' in regular expressions, which create deferment trees with lots of structural similarity. Experimental results show that, a DFA with 25000 states occupies only 0.59 Mb TCAM space for entry storage with optimizations of transition sharing and table consolidation.

Meiners further appended variable striding mechanism, thus to consume multiple characters per TCAM lookup. However, this will definitely increase the space requirement for the memory-scare TCAMs. For eight DFAs varies from 6533 states to 13825 states, they reported a throughput between 10 and 19 Gbps on a TCAM chip of 2.36 Mb.

The above entry merging algorithms are compression mechanism of DFA state transitions in nature, and they are effective for small scale DFAs. However, even they can achieve a approximately linear compression, it is still invalid when encountering large and complex rule sets, where DFA generation is even infeasible. As described in Section IV-A, state explosion is mainly caused by '.∗' and large character class repetitions. Peng *et al.* [152] leveraged the relationships between NFA states and DFA states to devise a two-segment state encoding mechanism which can efficiently inhibit state inflation. The two-segment state encoding method can efficiently merge DFA states that originate from the same NFA states, and it mainly aims at state replication caused by '.∗'. Experimental results show that, for Snort pattern sets with DFA size ranging from 13825 to 190951, the deflation method achieves up to two orders of magnitude lower for DFA size and TCAM entries. However, this method depends on computing the relationships between DFA states and NFA states, thus it cannot avoid the time-consuming DFA generation process.

As for another origination of state explosion, namely the wide character class with counting constraint, Meiners *et al.* [154] proposed to employ the *counting*-DFA [118] to handle the counters. *Counting*-DFA utilizes extra scratch memory as counters, thus to prevent the quadratic or even exponential expansion of counting constraint. Regardless of the limitations of *counting*-DFA analyzed in Section IV-C, the complex processing of counters will result in a sharp decrement in matching performance.

Huang *et al.* [179] proposed a TCAM-based matching automaton called CFA, to solve the scalable problem for large and complex rule sets. In fact, CFA is a concise representation of XFA. Considering the transition expansion problem in XFA, Huang proposed three compression techniques, transition compression, character compression and state compression. They are actually similar with optimizations proposed by Meiners *et al.* [153], experimental results reported that for a Bro set with 3644 DFA states, CFA reduces up to 83% of TCAM space and power consumption compared to optimized DFA and achieves a throughput up to 10.9 Gbps. For Bro sets with hundreds of thousands of DFA states, the reduction ratio can reach up to 95%. With assistance of multi-stride mechanism, the throughput can also reach up to several Gbps.

In a nutshell, TCAM is suit for pattern matching for its powerful parallel searching and high-speed memory access. String matching has achieved great improvement on TCAM, but it is still a great challenge to achieve satisfied performance for regular expressions. DFA is the most efficient representation for matching, but the space of TCAM severely limits the scale of DFA, and DFA is even infeasible when encountering the practical large and complex rule sets. Thus, the general multi-stride method for improving RE matching speed is unpractical in practice. In addition, TCAM also suffers high cost and high power consumption. TCAMs cost about 30 times more money per bit than DDR SRAMs, and consume 150 times more power per bit than SRAMs [83].

VI. Guidelines for Building Efficient Pattern Matching Components

In this section, we provide guidelines of building efficient pattern matching components for DPI applications. As we have declared, we would like to focus on the pattern matching process rather than a whole DPI system or a specific DPI application. But for supplementary, we also recommend that readers may refer to [54] for instructions to avoid packet loss when capturing packets on network cards and to accelerate packet transfer with improved operating system. In addition, the design of a specific DPI system should be considered from a global view in practise. As we have discussed in Section V-C, to fully exploit the potential global parallelism, we should divide the high level network analysis into multiple stages and estimate the potential parallel degree for each step, thus we can balance the throughput for each stage and achieve a better overall performance.

When focusing on the pattern matching process, we prefer to discuss it from two aspects, namely building efficient automatons and exploiting parallel platforms. The automatons are the core components for pattern matching, they provide compact and efficient structures for the signature sets, while the hardware platforms extract the potential parallelism for the matching mechanisms. Though there exists some relationship between the automatons and lower level platforms to some extent, they are still very independent in general as most automatons can be implemented in any platform. Thus, we can discuss them separately.

A. Guidelines for Building Efficient Automata

In this subsection, we present guidelines for building efficient automata in different applications and different levels of pattern scale and complexity. The presentation starts from small and simple pattern set to the challenging large and complex pattern set.

For exact string matching applications, like the string patterns in antivirus system ClamAV [176] and Web content filtering system DansGuardian [180], the original DFA is the most appropriate choice as DFA scale is linear with pattern length. Thus the popular platforms can support thousands of patterns even in fast memories, and any input character only requires one memory access. We strongly recommend the open source RegEx Processor [93] as the compiler for both NFA and DFA generation. RegEx Processor optimizes the original NFA to minimize the number of states and the number of transitions, for NFA to DFA conversion, it employs the prefix tree to significantly reduce the compilation complexity. In addition, RegEx Processor also provides transition diagram file for DFA, which can be displayed by the open source graph visualization software Graphviz [181]. Furthermore, if the DFA scale is small enough, multi-stride DFA can be employed to further improve the matching efficiency. Multi-stride DFA can read and process multiple characters at the same time, but the memory requirement of primitive multi-stride DFA is several orders of magnitude greater than original DFA, thus only works well for very small scale DFA.

For small-scale RE sets or the mixture with exact string patterns, the original DFA is also appropriate. But when the memory requirement exceeds the limit of fast memories, the automaton will be deployed on large but slow memories, like DRAM, which may lead to a rapid decline in performance. If the DFA scale is not more than one order of the amount of fast memories, the compression algorithm in Section IV-B can be adopted to reduce the memory requirement as most compression algorithms can achieve a compression ratio of more than 90%. Then, the compressed FA can be loaded into fast memories for rapid memory access. On the other hand, the compressed FA will increase the memory access times, thus the compression algorithm should be selected prudently to achieve a better performance. We do not recommend state merging algorithms and alphabet re-encoding algorithms as they only work well for smaller DFAs, the STT compression algorithms like D^2FA [100], the advanced version CD^2FA [101] and $A-DFA$ [106] , $FECAN$ [110], σFA [103] are good choices. RCDFA [79], [109] is a superior model, as it also optimizes the matching process to maintain comparable matching complexity to the original DFA.

Except for the traditional compression methods, another strategy which leverages the access model of state transition table is worthy of mentioning. In this way, we utilize the original state transition table of DFA rather than compress it. In general, the accesses to a given DFA will exhibit the following scenario. A few states have very high access frequency, while most states have very low access frequency or even no access. With this access model, we can deploy these few states in fast memories and other states in low large memories to achieve an overall better performance. Readers may refer to [182], where Chen employs Markov model [183] to compute the access probabilities.

With the continued growth in pattern scale or complexity, the DFA maybe too huge to compress, or the DFA is even infeasible in many situations like Snort and L7-filter. In these cases, the compressed FAs are not good choices due to the huge memory requirement. Then, NFA and scalable FAs are the remaining choices. NFA is not advised for its bad and unstable performance. Rule grouping can achieve a relatively stable performance due to its regular data structures, but it is not a wise choice for applications that will frequently update the patterns like NIDS, because the construction time is very long as discussed before. Even with the recently proposed estimating method [97], the computation time is still unacceptable for fast update requirements. In addition, rule grouping is limited to the unsupported pattern like "*AUTH\s*[^\n]{100}". In fact, rule grouping is very suitable for parallel platforms as different rule sets can be distributed to different engines for parallel processing. In practice, we strongly recommend Semi-determined FAs especially Becchi and Crowley's Hybrid-FA [95] for NIDS or other network security applications. As most network streams are benign, these packets cannot match the malicious signatures, they are not even similar with them. In other words, most streams will keep running in the DFA part and never advance to the NFA part, thus can achieve a relatively high performance on the whole. On the contrary, semi-determined FAs are not recommended for traffic classification applications, as most streams would match some traffic signatures. Decomposed FAs replace states with auxiliary variables and instructions, thus they can compile very complex patterns which maybe challenging for rule grouping FAs or semi-determined FAs. However, the performance maybe not very good due to the massive instruction fetching and variable calculation. Recently published researches [117], [119] have made some improvements for this issue, and make them more attractive for very complex pattern sets.

B. Guidelines to Utilize Parallel Platforms Efficiently

Among all the parallel platforms, FPGA is the only circuit based mechanism, and it is most suitable for NFA implementation. Here, we first provide several guidelines for implementing regular expression matching on FPGA platforms. First of all, updating speed is the major limitation of FPGA platforms as it involves time-consuming resynthesis of the circuits. Any changes in the rule set requires the recompilation of the automaton, resynthesis, replacement and routing of the circuits [133]. Thus, it is not very suitable for scenarios where signatures are altered frequently like NIDS or virus detection system. In general, the implementation includes three steps: converting regular expressions to corresponding NFA, NFA reduction and mapping NFA to circuits. NFA generation can refer to the standard compilation algorithms [84]–[86]. NFA reduction is very crucial in FPGA implementation, because NFA reduction not only reduces the required LUTs and FlipFlops to support more regular expressions, but also provides higher chip frequency and lower circuit delay, which can result in a higher matching throughput. The NFA reduction involves both state reduction and transition reduction, thus there needs a balance between these two contradictory factors in practice. For NFA reduction, readers may refer to [129] for prefix sharing and [122], [132] for infix sharing and suffix sharing. In addition, complex syntaxes like character class with counting constraints would occupy large hardware resource in traditional mapping mechanism. This can be improved with counter block approaches proposed in [130] and [131]. In addition, if the NFA is small or simple enough, multi-stride NFA [125], [126] can be employed to further improve the matching throughput.

Except for the traditional circuit-based automaton architecture, FPGA can also be configured as a memory based architecture, which means that the FPGA circuits are compiled to matching engines and the fast on-chip bank memories are utilized to store the DFA state transition table. Chen and Lu [182] provided a typical implementation on this architecture, where the author employs two-level memories for transition table storage. The first-level memory leverages the fast on-chip memory banks to cache the frequently accessed states (also called hot states), and the second-level large DRAM is used to hold the whole transition table of DFA, thus to achieve a high overall performance. With 9 matching engines running independently and parallelly, the performance reaches to 29.6 Gbps on Altera Stratix II EP2S 180 with moderate DFA.

ASICs have high area/energy and speed, but they also have high system design cost and poor flexibility, thus they only suit for large scale industry production not for common researches. For other memory based parallel platforms, SIMD processors, mainly GPUs have the most abundant parallel execution units. Thus they can perform better than the superscalar network processors and general purpose multi-core processors in general, and current high throughputs in experiments are most achieved on GPU platforms. However, they are very sensitive to irregular memory accesses and divergences in execution path. Thus they are more suitable for DFA or DFA-like automatons, which have more regular access model and less divergences. On the other hand, network processors and general purpose multi-core processors have much more hardware resources for each execution unit, so they can sustain more complex access models and divergences, meaning that they are more suitable for more complex automatons.

GPU is usually equipped as a coprocessor in DPI systems, and the programming model is a little complex for novices. Here we list some practical guidelines for building DPI systems with GPUs. The guidelines mainly include packet transferring from CPU to GPU, the set of packet buffer size, the choice for matching automatons, the deployment of packets and automatons on GPU memory, reading packets from GPU memory to threads, the parallel matching process, and other optimizing suggestions.

GPU cannot read packets directly from network interface, thus GPU needs to read the packets in CPU memory via PCIe bus, which connects the GPU card with host system. Though multi-lane PCIe bus can reach the peak performance of tens of Gbps, it may suffer large overhead when encountering small packet transfers, dropping to only a few several Gbps. Thus, it is preferable to copy the packets to GPU in batches rather than transfer them separately. A separate buffer in CPU memory can be allocated for holding the packets temporarily, whenever the buffer gets full, the packets can be transferred to GPU in a batch. For further improvement, the buffer memory type can be configured as page-locked memory, which is kind of memory maps no virtual address and can not be swapped out from the main memory. In addition, DMA technique can be employed for the transfer from page-locked memory to GPU, thus releasing the CPU for other works. Double buffering technique is also recommended for page-locked memory, when a buffer is full and copied to GPU, the other buffer is still available for CPU to store the new coming packets.

Packet buffer size is also a design parameter that needs to be considered. Vasiliadis *et al.* [133] has explored the impact of buffer size on the overall system performance, including the packet capture, decoding, classification and transferring. Results show that, the transfer cost decreases with the incremental buffer size, and remains stable for a specific buffer size. As the optimal buffer size is related to the specific devices, extra experiments are required to determine the optimal value. It a little complex when TCP stream reassembly is required in some applications, especially in modern NDISs, where packets from same from are aggregated into a single stream to inspect signatures across multiple packets. Then, the buffer organization should consider the reassembly factor. While this is out of our matching scope, readers may refer to related researches for deep understanding.

The choice of finite automaton is a main issue in practice. The original DFA is most suitable to GPU architectures for its regular data structure and matching process, only one state needs to be traversed for each input character, independent of the pattern set and trace. Thus, as long as the DFA scale is not exceeding the space of GPU global memory, DFA is the first choice. Otherwise, other kind of FAs like compressed DFAs or NFA-like automatons can be employed for pattern matching. For large and complex rule sets, the iNFAnt and its enhanced version are strongly recommend for the following reasons. First, as iNFAnt is NFA based automaton, thus can scale to large and complex pattern sets without state explosion. Second, the symbol-first representation utilizes much of the available bandwidth, and omits much divergence in traditional NFA. In a nutshell, the basic principle is to avoid irregular representations, thus to avoid divergences.

Following are the deployments of automatons and packets on GPU memories. As faster memories always result in higher performance, it is better to deploy them on faster memories, like texture memories as they are cached on chip. A hit in cache would reduce the memory access latency from hundreds of cycles to several cycles, bringing a big improvement for overall performance. For older GPU architectures, texture memory is a good choice if the space is sufficient. For new GPU architectures like Fermi architecture, global memory is also cached on chip and performs as well as texture memory. Then the deployment would be more flexible, users may determine a optimal scheme with aforehand experiments. In addition, software-managed fast on-chip memory can also help to reduce memory access latencies [137], like the shared memory of NVIDIA GPUs, and the stream register file of Imagin [184]. As we have discussed before, some hot states can be chose to deployed on these on-chip memories through profiling the rule sets and traces.

The next step is to fetch the packet bytes from GPU texture or global memory to the matching threads. Experiments [136] show that reducing the number of memory accesses can bring a significant improvement in the overall performance. Due to the regular and sequential access to packet content, it is beneficial to fetch multiple bytes one time. In general, each packet is assigned to a thread for matching, then how many threads should be set to achieve the best performance? Vasiliadis *et al.* [136] argued that it is related to the internal GPU thread scheduler, thus experiments should be conducted aforehand to determine the optimal thread number.

Strictly speaking, TCAM cannot be called an execution platform, it is a kind of memory. Although most modern network devices have equipped TCAM, it is impractical to implement the pattern matching process with the whole TCAM, because most of its space has been used for fast routing and flow table management. In addition, TCAM is usually very expensive and energy-consuming. Thus, it is recommended that we can leverage residual TCAM resource for auxiliary processing.

VII. Conclusion

Regular expression matching is a core component for many deep packet inspection applications, where the packet payload should be inspected against a set of patterns. In this paper, we have comprehensively surveyed the issues for regular expression matching of deep packet inspection from a systematic perspective. Sufficient application background as well as technical background are provided for deep packet inspection, thus readers from general fields can achieve a global and deep understanding for DPI quickly. We explained how regular expression is used for DPI and what are the challenges in practice. Except for hardware cost, power consumption and fast update, the biggest challenge is to fulfill the throughput requirement under conditions of rapid growing link speed and pattern sets with current hardware platforms. Automaton based regular expression matching is a state traversal process driven by the payload bytes, each time the matching engine reads a payload byte, and inquires the automaton for the next state(s). As the automaton is deployed in memories, matching speed is mainly determined by the memory access latencies.

Large and complex pattern set results in exponential growing memory requirement which is several orders more than the capacity of modern high speed memories. While, the speed that big-capacity memories can provide is several orders lower than the link speed. Thus, the main goal is to minimize the storage requirement with not much sacrifice in performance. Current researches can be classified to software solutions and hardware solutions roughly. Software solutions solve state expansion problems through exploiting compact representations for finite automatons, while the cost is more irregular automatons which require more memory access operations or computations when matching. Hardware solutions leverage massive hardware parallelism to accelerate matching throughput. However, most hardware platforms have inherent shortages which cannot fully satisfy the requirement for DPI. We also provide guidelines for building practical automatons for different scenarios, as well as how to efficiently leverage hardware platforms, like FPGAs and GPUs.

The current credible throughput for large complex pattern sets is not more than 1 Gbps. To keep up with the growing link speed, there still needs a lot of innovative researches from both hardware and software. Both the convenience and troubles come from the powerful expressive ability of regular expression. It is also necessary to exploit other kinds of expressions in theory. In addition, sophisticated FAs may bring considerable memory reduction and performance improvement, but they must be combined with specific hardware circuits, thus innovation from hardware architecture is also essential to support high speed REM. It is also a new direction to combine different devices on the same chip or board to fully utilize their advantages. Furthermore, by guaranteeing a defined error probability, the system may require less hardware resource and achieve a better performance. This is especially useful for NIDS applications, as whether generated manually or automatically, practical patterns brings intrinsic false positives and false negatives in the match detection [82].

In addition, pattern matching can be combined with other techniques like heavy hitter, port scan and change detection for NIDS applications [82], because pattern matching is applicable only when the features of malicious traffic are known and the packet payload is not encrypted. Becchi *et al.* [185] has advanced to the direction of accelerating regular expression matching over compressed HTTP. Techniques in regular expression such as memory compression algorithms and protocol classification can also be extended to other areas, like computational linguistics and structured data parsing.

Acknowledgment

The authors are grateful to the editors and reviewers for their kind and valuable comments that greatly improve the quality of the paper.

References

[1] A. V. Aho and M. J. Corasick, "Efficient string matching: An aid to bibliographic search," *Commun. ACM*, vol. 18, no. 6, pp. 333–340, 1975.
[2] S. Wu and U. Manber, "A fast algorithm for multi-pattern searching," Dept. Comput. Sci., Univ. Arizona, Tucson, AZ, USA, Tech. Rep. TR-94-17, 1994.
[3] C. Allauzen and M. Raffinot, "Factor oracle of a set of words," Institute Gaspart-Monge, University de Marne-la-vallee, Champs-sur-Marne, France, Tech. Rep. TR-99-11, 1999.
[4] *Snort v2.9.* (2014). [Online]. Available: http://www.snort.org/
[5] *Bro Intrusion Detection System.* (2014). [Online]. Available: http://www.bro.org/
[6] *Application Layer Packet Classifier for LINUX.* (2009). [Online]. Available: http://l7-filter.sourceforge.net/
[7] *Cisco IOS IPS Deployment Guide.* Accessed on Jun. 10, 2015. [Online]. Available: http://www.cisco.com/
[8] *Cavium, OCTEON5860.* Accessed on Jun. 10, 2015. [Online]. Available: http://www.cavium.com/OCTEON_MIPS64.html/
[9] *IBM PowerEN PME Public Pattern Sets WiKi.* (2012). [Online]. Available: https://www.ibm.com/developerworks/mydeveloperworks/wikis/home?lang=en#/wiki/PowerEN%20PME%20Public%20Pattern%20Sets/page/Welcome/
[10] *Deep Packet Inspection From Wikipedia.* Accessed on Jun. 27, 2015. [Online]. Available: https://en.wikipedia.org/wiki/Deep_packet_inspection/
[11] M. Mueller, "DPI technology from the standpoint of Internet governance studies: An introduction," School Inf. Studies, Syracuse Univ., Syracuse, NY, USA, Tech. Rep., 2011. [Online]. Available: http://dpi.ischool.syr.edu/Technology_files/WhatisDPI-2.pdf
[12] K. Mochalski and H. Schulze, "Deep packet inspection: Technology, applications & net neutrality," White Paper, 2009, pp. 1–12.
[13] R. Bendrath and M. Mueller, "The end of the net as we know it? Deep packet inspection and Internet governance," *New Media Soc.*, vol. 13, no. 7, pp. 1142–1160, 2011.

[14] R. Bendrath, "Global technology trends and national regulation: Explaining variation in the governance of deep packet inspection," in *Proc. Int. Studies Annu. Conv.*, New York, NY, USA, Feb. 2009, pp. 15–18.

[15] I. Sourdis, "Designs and algorithms for packet and content inspection," Ph.D. dissertation, TU Delft, Delft Univ. Technol., Delft, The Netherlands, 2007.

[16] K. Haley, "Internet security threat report," Symantec, Mountain View, CA, USA, 2015. [Online]. Available: https://www4.symantec.com/mktginfo/whitepaper/ISTR/21347931_GA-internet-security-threat-report-volume-20-2015-appendices.pdf/

[17] *Pandalabs Report Q1 2015*. (2015). [Online]. Available: http://www.pandasecurity.com/mediacenter/src/uploads/2015/05/PandaLabs-Report_Q1-2015.pdf/

[18] M. Wedel and A. Roessler, "Data loss prevention," U.S. Patent 7 185 238, Feb. 27, 2007.

[19] K. Ruan, J. Carthy, T. Kechadi, and M. Crosbie, "Cloud forensics," in *Advances in Digital Forensics VII*. Heidelberg, Germany: Springer, 2011, pp. 35–46.

[20] H. Asghari, M. Van Eeten, J. M. Bauer, and M. Mueller, "Deep packet inspection: Effects of regulation on its deployment by Internet providers," in *Proc. 41st Res. Conf. Commun. Inf. Internet Policy*, Arlington, VA, USA, Sep. 2013. .

[21] C.-S. Moon and S.-H. Kim, "A study on the integrated security system based real-time network packet deep inspection," *Int. J. Security Appl.*, vol. 8, no. 1, pp. 113–122, 2014.

[22] *Cisco Visual Networking Index: Forecast and Methodology, 2014-2019*. (2015). [Online]. Available: http://www.cisco.com/c/en/us/solutions/collateral/service-provider/ip-ngn-ip-next-generation-network/white_paper_c11-481360.html

[23] *Advanced DPI: Intelligent Security and Insight Add Up to Opportunity, Strategic White Paper, Alcatel Lucent*. (2008). [Online]. Available: http://www.bitpipe.com/detail/RES/1201637172_856.html

[24] K. David. (2013). *Ad-Injecting Trojan Targets Mac Users on Safari, Firefox, and Chrome*. [Online]. Available: http://arstechnica.com/apple/2013/03/

[25] K. Thomas *et al.*, "Ad injection at scale: Assessing deceptive advertisement modifications," in *Proc. IEEE Security Privacy*, San Jose, CA, USA, 2015, pp. 151–167.

[26] M. Mueller, A. Kuehn, and S. M. Santoso, "Policing the network: Using dpi for copyright enforcement," *Surveillance Soc.*, vol. 9, no. 4, pp. 348–364, 2012.

[27] *Automatic Content Recognition Content Identification Solutions Trusted by Industry Leaders*. (2015). [Online]. http://www.audiblemagic.com/index.php?m=view&key=13

[28] *Prism (Surveillance Program), From Wikipedia, the Free Encyclopedia*. (2015). [Online]. Available: https://en.wikipedia.org/wiki/PRISM_(surveillance_program)

[29] M. Finsterbusch, C. Richter, E. Rocha, J.-A. Muller, and K. Hanssgen, "A survey of payload-based traffic classification approaches," *IEEE Commun. Surveys Tuts.*, vol. 16, no. 2, pp. 1135–1156, 2nd Quart. 2014.

[30] P.-C. Lin, Z.-X. Li, Y.-D. Lin, Y.-C. Lai, and F. C. Lin, "Profiling and accelerating string matching algorithms in three network content security applications," *IEEE Commun. Surveys Tuts.*, vol. 8, no. 2, pp. 24–36, 2nd Quart. 2006.

[31] P.-C. Lin, Y.-D. Lin, Y.-C. Lai, and T.-H. Lee, "Using string matching for deep packet inspection," *Computer*, vol. 41, no. 4, pp. 23–28, Apr. 2008.

[32] *Service Name and Transport Protocol Port Number Registry*. (2015). [Online]. http://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml

[33] Y. Qi, L. Xu, B. Yang, Y. Xue, and J. Li, "Packet classification algorithms: From theory to practice," in *Proc. IEEE INFOCOM*, Rio de Janeiro, Brazil, 2009, pp. 648–656.

[34] A. W. Moore and K. Papagiannaki, "Toward the accurate identification of network applications," in *Passive and Active Network Measurement*. Heidelberg, Germany: Springer, 2005, pp. 41–54.

[35] A. Madhukar and C. Williamson, "A longitudinal study of P2P traffic classification," in *Proc. 14th IEEE Int. Symp. Model. Anal. Simulat. Comput. Telecommun. Syst. (MASCOTS)*, Monterey, CA, USA, 2006, pp. 179–188.

[36] M. Dusi, F. Gringoli, and L. Salgarelli, "Quantifying the accuracy of the ground truth associated with Internet traffic traces," *Comput. Netw.*, vol. 55, no. 5, pp. 1158–1167, 2011.

[37] P. Piskac and J. Novotny, "Using of time characteristics in data flow for traffic classification," in *Managing the Dynamics of Networks and Services*. Heidelberg, Germany: Springer, 2011, pp. 173–176.

[38] J. Y. Chung, B. Park, Y. J. Won, J. Strassner, and J. W. Hong, "Traffic classification based on flow similarity," in *IP Operations and Management*. Heidelberg, Germany: Springer, 2009, pp. 65–77.

[39] E. Rocha, P. Salvador, and A. Nogueira, "Detection of illicit network activities based on multivariate Gaussian fitting of multi-scale traffic characteristics," in *Proc. IEEE Int. Conf. Commun. (ICC)*, Kyoto, Japan, 2011, pp. 1–6.

[40] T. T. T. Nguyen and G. Armitage, "A survey of techniques for Internet traffic classification using machine learning," *IEEE Commun. Surveys Tuts.*, vol. 10, no. 4, pp. 56–76, 4th Quart. 2008.

[41] F. G. O. Risso, M. Baldi, O. Morandi, A. Baldini, and P. Monclus, "Lightweight, payload-based traffic classification: An experimental evaluation," in *Proc. IEEE Int. Conf. Commun.*, Beijing, China, 2008, pp. 5869–5875.

[42] R. S. Boyer and J. S. Moore, "A fast string searching algorithm," *Commun. ACM*, vol. 20, no. 10, pp. 762–772, 1977.

[43] A. C.-C. Yao, "The complexity of pattern matching for a random string," *SIAM J. Comput.*, vol. 8, no. 3, pp. 368–387, 1979.

[44] Z. Galil, "On improving the worst case running time of the Boyer–Moore string matching algorithm," *Commun. ACM*, vol. 22, no. 9, pp. 505–508, 1979.

[45] R. N. Horspool, "Practical fast searching in strings," *Softw. Pract. Experience*, vol. 10, no. 6, pp. 501–506, 1980.

[46] G. Navarro, "Flexible pattern matching," *J. Appl. Stat.*, vol. 31, 2002.

[47] 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," *ACM Trans. Embedded Comput. Syst.*, vol. 3, no. 3, pp. 614–633, 2004.

[48] R. Muth and U. Manber, "Approximate multiple string search," in *Combinatorial Pattern Matching*. Heidelberg, Germany: Springer, 1996, pp. 75–86.

[49] R. M. Karp and M. O. Rabin, "Efficient randomized pattern-matching algorithms," *IBM J. Res. Develop.*, vol. 31, no. 2, pp. 249–260, 1987.

[50] M. Norton, *Optimizing Pattern Matching for Intrusion Detection*, Sourcefire, Inc., Columbia, MD, USA, 2004.

[51] N. Tuck, T. Sherwood, B. Calder, and G. Varghese, "Deterministic memory-efficient string matching algorithms for intrusion detection," in *Proc. 23rd Annu. Joint Conf. IEEE Comput. Commun. Soc. (INFOCOM)*, vol. 4. Hong Kong, 2004, pp. 2628–2639.

[52] L. Tan and T. Sherwood, "A high throughput string matching architecture for intrusion detection and prevention," in *Proc. ACM SIGARCH Comput. Archit. News*, vol. 33, no. 2, 2005, pp. 112–122.

[53] A. Callado *et al.*, "A survey on Internet traffic identification," *IEEE Commun. Surveys Tuts.*, vol. 11, no. 3, pp. 37–52, 3rd Quart. 2009.

[54] R. Antonello *et al.*, "Deep packet inspection tools and techniques in commodity platforms: Challenges and trends," *J. Netw. Comput. Appl.*, vol. 35, no. 6, pp. 1863–1878, 2012.

[55] *The Libpcap Project*. (2015). [Online]. Available: http://sourceforge.net/projects/libpcap/

[56] *LibpcapMMAP*. (2012). [Online]. Available: http://public.lanl.gov/cpw/

[57] *Pf_Ring:HighSpeed Packet Capture, Filtering and Analysis*. (2015). [Online]. Available: http://www.ntop.org/products/packetcapture/pf_ring/

[58] *UserSpace e1000 Driver Library*. (2015). [Online]. Available: http://sourceforge.net/projects/libe1000/

[59] N. Bonelli, A. Di Pietro, S. Giordano, and G. Procissi, "On multi–gigabit packet capturing with multi–core commodity hardware," in *Passive and Active Measurement*. Heidelberg, Germany: Springer, 2012, pp. 64–73.

[60] L. Rizzo, "Netmap: A novel framework for fast packet I/O," in *Proc. USENIX Annu. Tech. Conf.*, Boston, MA, USA, 2012, pp. 101–112.

[61] L. Rizzo, L. Deri, and A. Cardigliano. (2012). *10 gbit/s Line Rate Packet Processing Using Commodity Hardware: Survey and New Proposals*. [Online]. Available: http://luca.ntop.org/10g.pdf

[62] N. Kim, G. Choi, and J. Choi, "A scalable carrier-grade DPI system architecture using synchronization of flow information," *IEEE J. Sel. Areas Commun.*, vol. 32, no. 10, pp. 1834–1848, Oct. 2014.

[63] *DPDK: Data Plane Development Kit*. (2015). [Online]. Available: http://dpdk.org/

[64] F. Schneider, J. Wallerich, and A. Feldmann, "Packet capture in 10Gigabit Ethernet environments using contemporary commodity hardware," in *Passive and Active Network Measurement*. Heidelberg, Germany: Springer, 2007, pp. 207–217.

[65] L. Deri *et al.*, "Improving passive packet capture: Beyond device polling," in *Proc. SANE*, Amsterdam, The Netherlands, 2004, pp. 85–93. [Online]. Available: http://luca.ntop.org/Ring.pdf

[66] S. Alcock, P. Lorier, and R. Nelson, "Libtrace: A packet capture and analysis library," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 42, no. 2, pp. 42–48, 2012.

[67] R. Hofstede *et al.*, "Flow monitoring explained: From packet capture to data analysis with NetFlow and IPFIX," *IEEE Commun. Surveys Tuts.*, vol. 16, no. 4, pp. 2037–2064, 4th Quart. 2014.

[68] *Ping of Death*. (2015). [Online]. Available: http://insecure.org/sploits/ping-o-death.html

[69] *Teardrop IP Fragmentation (Teardrop)*. (2015). [Online]. http://www.iss.net/security_center/reference/vuln/TearDrop.htm

[70] S. Egorov and G. Savchuk, "SNORTRAN: An optimizing compiler for snort rules," *Fidelis Security Systems*, 2002. [Online]. Available: https://dl.packetstormsecurity.net/papers/IDS/SNORTRAN-wp.pdf

[71] S. Chen, R. Lu, and X. S. Shen, "SRC: A multicore NPU-based TCP stream reassembly card for deep packet inspection," *Security Commun. Netw.*, vol. 7, no. 2, pp. 265–278, 2014.

[72] *Libnids*. (2010). [Online]. Available: http://libnids.sourceforge.net/

[73] *TCPflow*. (2015). [Online]. Available: https://github.com/simsong/tcpflow

[74] G. Wagener, A. Dulaunoy, and T. Engel, "Towards an estimation of the accuracy of TCP reassembly in network forensics," in *Proc. 2nd Int. Conf. Future Gener. Commun. Netw. (FGCN)*, vol. 2. 2008, pp. 273–278.

[75] M. Zhang and J.-B. Ju, "Space-economical reassembly for intrusion detection system," in *Information and Communications Security*. Heidelberg, Germany: Springer, 2003, pp. 393–404.

[76] S. Dharmapurikar and V. Paxson, "Robust TCP stream reassembly in the presence of adversaries," in *Proc. USENIX Security*, Baltimore, MD, USA, 2005, pp. 65–80.

[77] T. AbuHmed, A. Mohaisen, and D. Nyang, "Deep packet inspection for intrusion detection systems: A survey," *Korea Commun. Soc. (Inf. Commun.)*, vol. 24, no. 11, pp. 25–36, 2007.

[78] P. M. Rathod, N. Marathe, and A. V. Vidhate, "A survey on finite automata based pattern matching techniques for network intrusion detection system (NIDS)," in *Proc. Int. Conf. Adv. Electron. Comput. Commun. (ICAECC)*, Bengaluru, India, 2014, pp. 1–5.

[79] R. Antonello, S. Fernandes, D. Sadok, J. Kelner, and G. Szabo, "Deterministic finite automaton for scalable traffic identification: The power of compressing by range," in *Proc. IEEE Netw. Oper. Manag. Symp. (NOMS)*, 2012, pp. 155–162.

[80] S. C. Kleene, "Representation of events in nerve nets and finite automata," RAND Corporat., Tech. Rep. RM 107, Santa Monica, CA, USA, 1951.

[81] M. A. Harrison, *Introduction to Formal Language Theory*. Reading, MA, USA: Addison-Wesley, 1978.

[82] M. Becchi, "Data structures, algorithms and architectures for efficient regular expression evaluation," Ph.D. dissertation, Dept. Comput. Sci. Eng., Washington Univ., St. Louis, MO, USA, 2009.

[83] F. Yu, "High speed deep packet inspection with hardware support," Ph.D. dissertation, Dept. Comput. Sci., Univ. California at Berkeley, Berkeley, CA, USA, 2006.

[84] R. McNaughton and H. Yamada, "Regular expressions and state graphs for automata," *IEEE Trans. Electron. Comput.*, vol. EC-9, no. 1, pp. 39–47, Mar. 1960.

[85] K. Thompson, "Programming techniques: Regular expression search algorithm," *Commun. ACM*, vol. 11, no. 6, pp. 419–422, 1968.

[86] J. E. Hopcroft, *Introduction to Automata Theory, Languages, and Computation*. Harlow, U.K.: Pearson Edu., 1979.

[87] M. Becchi and P. Crowley, "Efficient regular expression evaluation: Theory to practice," in *Proc. 4th ACM/IEEE Symp. Archit. Netw. Commun. Syst.*, San Jose, CA, USA, 2008, pp. 50–59.

[88] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers, Principles, Techniques and Tools*. Reading, MA, USA: Addison-Wesley, 1986.

[89] J. Berstel, L. Boasson, O. Carton, and I. Fagnot, "Minimization of automata," *arXiv preprint arXiv:1010.5318*, 2010. [Online]. Available: http://arxiv.org/pdf/1010.5318.pdf

[90] J. van Lunteren, "High-performance pattern-matching for intrusion detection," in *Proc. INFOCOM*, vol. 6. Barcelona, Spain, Apr. 2006, pp. 1–13.

[91] K. Atasu, R. Polig, J. Rohrer, and C. Hagleitner, "Exploring the design space of programmable regular expression matching accelerators," *J. Syst. Archit.*, vol. 59, no. 10, pp. 1184–1196, 2013.

[92] X. Wang *et al.*, "Kangaroo: Accelerating string matching by running multiple collaborative finite state machines," *IEEE J. Sel. Areas Commun.*, vol. 32, no. 10, pp. 1784–1796, Oct. 2014.

[93] *Regular Expression Processor*. Accessed on Apr. 5, 2012. [Online]. Available: http://regex.wustl.edu/index.php/Regular_Expression_Processor

[94] F. Yu, Z. Chen, Y. Diao, T. V. Lakshman, and R. H. Katz, "Fast and memory-efficient regular expression matching for deep packet inspection," in *Proc. ACM/IEEE Symp. Archit. Netw. Commun. Syst.*, San Jose, CA, USA, 2006, pp. 93–102.

[95] M. Becchi and P. Crowley, "A hybrid finite automaton for practical deep packet inspection," in *Proc. ACM CoNEXT Conf.*, New York, NY, USA, 2007, Art. no. 1.

[96] Y.-H. E. Yang and V. K. Prasanna, "Space-time tradeoff in regular expression matching with semi-deterministic finite automata," in *Proc. IEEE INFOCOM*, Shanghai, China, 2011, pp. 1853–1861.

[97] T. Liu, A. X. Liu, J. Shi, Y. Sun, and L. Guo, "Towards fast and optimal grouping of regular expressions via DFA size estimation," *IEEE J. Sel. Areas Commun.*, vol. 32, no. 10, pp. 1797–1809, Oct. 2014.

[98] M. Becchi and S. Cadambi, "Memory-efficient regular expression search using state merging," in *Proc. 26th IEEE Int. Conf. Comput. Commun. (INFOCOM)*, Anchorage, AK, USA, 2007, pp. 1064–1072.

[99] B. C. Brodie, D. E. Taylor, and R. K. Cytron, "A scalable architecture for high-throughput regular-expression pattern matching," in *Proc. ACM SIGARCH Comput. Archit. News*, vol. 34, no. 2, 2006, pp. 191–202.

[100] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner, "Algorithms to accelerate multiple regular expressions matching for deep packet inspection," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 36, no. 4, pp. 339–350, 2006.

[101] S. Kumar, J. Turner, and J. Williams, "Advanced algorithms for fast and scalable deep packet inspection," in *Proc. ACM/IEEE Symp. Architect. Netw. Commun. Syst.*, San Jose, CA, USA, 2006, pp. 81–92.

[102] M. Becchi and P. Crowley, "An improved algorithm to accelerate regular expression evaluation," in *Proc. 3rd ACM/IEEE Symp. Architect. Netw. Commun. Syst.*, Orlando, FL, USA, 2007, pp. 145–154.

[103] D. Ficara *et al.*, "An improved DFA for fast regular expression matching," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 5, pp. 29–40, 2008.

[104] D. Ficara *et al.*, "Differential encoding of DFAs for fast regular expression matching," *IEEE/ACM Trans. Netw.*, vol. 19, no. 3, pp. 683–694, Jun. 2011.

[105] S. Kong, R. Smith, and C. Estan, "Efficient signature matching with multiple alphabet compression tables," in *Proc. 4th Int. Conf. Security Privacy Commun. Netow.*, Istanbul, Turkey, 2008, Art. no. 1.

[106] M. Becchi and P. Crowley, "A-DFA: A time-and space-efficient DFA compression algorithm for fast regular expression evaluation," *ACM Trans. Architect. Code Optim. (TACO)*, vol. 10, no. 1, 2013, Art. no. 4.

[107] J. Patel, A. X. Liu, and E. Torng, "Bypassing space explosion in high-speed regular expression matching," *IEEE/ACM Trans. Netw.*, vol. 22, no. 6, pp. 1701–1714, Dec. 2014.

[108] A. X. Liu and E. Torng, "An overlay automata approach to regular expression matching," in *Proc. IEEE INFOCOM*, Toronto, ON, Canada, 2014, pp. 952–960.

[109] R. Antonello, S. Fernandes, D. Sadok, J. Kelner, and G. Szabó, "Design and optimizations for efficient regular expression matching in DPI systems," *Comput. Commun.*, vol. 61, pp. 103–120, May 2015.

[110] Y. Qi *et al.*, "Feacan: Front-end acceleration for content-aware network processing," in *Proc. IEEE INFOCOM*, Shanghai, China, 2011, pp. 2114–2122.

[111] J. Rohrer, K. Atasu, J. van Lunteren, and C. Hagleitner, "Memory-efficient distribution of regular expressions for fast deep packet inspection," in *Proc. 7th IEEE/ACM Int. Conf. Hardw. Softw. Codesign Syst. Synth.*, Grenoble, France, 2009, pp. 147–154.

[112] S. Kumar, B. Chandrasekaran, J. Turner, and G. Varghese, "Curing regular expressions matching algorithms from insomnia, amnesia, and acalculia," in *Proc. 3rd ACM/IEEE Symp. Architect. Netw. Commun. Syst.*, Orlando, FL, USA, 2007, pp. 155–164.

[113] R. Smith, C. Estan, S. Jha, and S. Kong, "Deflating the big bang: fast and scalable deep packet inspection with extended finite automata," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 4, pp. 207–218, 2008.

[114] R. Smith, C. Estan, and S. Jha, "XFA: Faster signature matching with extended automata," in *Proc. IEEE Symp. Security Privacy*, Oakland, CA, USA, 2008, pp. 187–201.

[115] K. Wang and J. Li, "Towards fast regular expression matching in practice," in *Proc. ACM SIGCOMM Conf.*, Hong Kong, 2013, pp. 531–532.

[116] Y. Xu, J. Jiang, R. Wei, Y. Song, and H. J. Chao, "TFA: A tunable finite automaton for pattern matching in network intrusion detection systems," *IEEE J. Sel. Areas Commun.*, vol. 32, no. 10, pp. 1810–1821, Oct. 2014.

[117] X. Yu, B. Lin, and M. Becchi, "Revisiting state blow-up: Automatically building augmented-fa while preserving functional equivalence," *IEEE J. Sel. Areas Commun.*, vol. 32, no. 10, pp. 1822–1833, Oct. 2014.

[118] M. Becchi and P. Crowley, "Extending finite automata to efficiently match Perl-compatible regular expressions," in *Proc. ACM CoNEXT Conf.*, Madrid, Spain, 2008, Art. no. 25.

[119] K. Wang, Z. Fu, X. Hu, and J. Li, "Practical regular expression matching free of scalability and performance barriers," *Comput. Commun.*, vol. 54, pp. 97–119, Dec. 2014.

[120] R. W. Floyd and J. D. Ullman, "The compilation of regular expressions into integrated circuits," *J. ACM (JACM)*, vol. 29, no. 3, pp. 603–622, 1982.

[121] R. Sidhu and V. K. Prasanna, "Fast regular expression matching using FPGAs," in *Proc. 9th Annu. IEEE Symp. Field Program. Custom Comput. Mach. (FCCM)*, Rohnert Park, CA, USA, 2001, pp. 227–238.

[122] C.-H. Lin, C.-T. Huang, C.-P. Jiang, and S.-C. Chang, "Optimization of pattern matching circuits for regular expression on FPGA," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 15, no. 12, pp. 1303–1310, Dec. 2007.

[123] Y.-H. E. Yang, W. Jiang, and V. K. Prasanna, "Compact architecture for high-throughput regular expression matching on FPGA," in *Proc. 4th ACM/IEEE Symp. Architect. Netw. Commun. Syst.*, San Jose, CA, USA, 2008, pp. 30–39.

[124] Y. H. Cho, S. Navab, and W. H. Mangione-Smith, "Specialized hardware for deep network packet filtering," in *Field-Programmable Logic and Applications: Reconfigurable Computing Is Going Mainstream*. Heidelberg, Germany: Springer, 2002, pp. 452–461.

[125] N. Yamagaki, R. Sidhu, and S. Kamiya, "High-speed regular expression matching engine using multi-character NFA," in *Proc. Int. Conf. Field Program. Logic Appl. (FPL)*, Heidelberg, Germany, 2008, pp. 131–136.

[126] H. Nakahara, T. Sasao, and M. Matsuura, "A regular expression matching circuit based on a modular non-deterministic finite automaton with multi-character transition," in *Proc. 16th Workshop Synth. Syst. Integr. Mixed Inf. technol.*, Taipei, Taiwan, 2010, pp. 359–364.

[127] V. Košar and J. Korenek, "Efficient mapping of nondeterministic automata to FPGA for fast regular expression matching," in *Proc. DDECS*, Vienna, Austria, 2010, pp. 54–59.

[128] V. Košař, M. Žádník, and J. Kořenek, "NFA reduction for regular expressions matching using FPGA," in *Proc. Int. Conf. Field Program. Technol. (FPT)*, Kyoto, Japan, 2013, pp. 338–341.

[129] J. Bispo, I. Sourdis, J. M. P. Cardoso, and S. Vassiliadis, "Regular expression matching for reconfigurable packet inspection," in *Proc. IEEE Int. Conf. Field Program. Technol. (FPT)*, Bangkok, Thailand, 2006, pp. 119–126.

[130] M. Faezipour and M. Nourani, "Constraint repetition inspection for regular expression on FPGA," in *Proc. 16th IEEE Symp. High Perform. Interconnects (HOTI)*, Stanford, CA, USA, 2008, pp. 111–118.

[131] H. Wang, S. Pu, G. Knezek, and J.-C. Liu, "Min-max: A counter-based algorithm for regular expression matching," *IEEE Trans. Parallel Distrib. Syst.*, vol. 24, no. 1, pp. 92–103, Jan. 2013.

[132] T. T. Hieu, T. N. Thinh, and S. Tomiyama, "ENREM: An efficient NFA-based regular expression matching engine on reconfigurable hardware for NIDS," *J. Syst. Architect.*, vol. 59, nos. 4–5, pp. 202–212, 2013.

[133] G. Vasiliadis, S. Antonatos, M. Polychronakis, E. P. Markatos, and S. Ioannidis, "Gnort: High performance network intrusion detection using graphics processors," in *Recent Advances in Intrusion Detection*. Heidelberg, Germany: Springer, 2008, pp. 116–134.

[134] G. Vasiliadis, M. Polychronakis, S. Antonatos, E. P. Markatos, and S. Ioannidis, "Regular expression matching on graphics hardware for intrusion detection," in *Recent Advances in Intrusion Detection*. Heidelberg, Germany: Springer, 2009, pp. 265–283.

[135] N. Cascarano, P. Rolando, F. Risso, and R. Sisto, "iNFAnt: NfA pattern matching on GPGPU devices," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 40, no. 5, pp. 20–26, 2010.

[136] G. Vasiliadis, M. Polychronakis, and S. Ioannidis, "Parallelization and characterization of pattern matching using GPUs," in *Proc. IEEE Int. Symp. Workload Characterization (IISWC)*, Austin, TX, USA, 2011, pp. 216–225.

[137] R. Smith, N. Goyal, J. Ormont, K. Sankaralingam, and C. Estan, "Evaluating GPUs for network packet signature matching," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw. (ISPASS)*, Boston, MA, USA, 2009, pp. 175–184.

[138] X. Yu and M. Becchi, "GPU acceleration of regular expression matching for large datasets: Exploring the implementation space," in *Proc. ACM Int. Conf. Comput. Front.*, Ischia, Italy, 2013, Art. no. 18.

[139] Y. Zu *et al.*, "GPU-based NFA implementation for memory efficient high speed regular expression matching," *ACM SIGPLAN Not.*, vol. 47, no. 8, pp. 129–140, 2012.

[140] C.-H. Lin, C.-H. Liu, L.-S. Chien, and S.-C. Chang, "Accelerating pattern matching using a novel parallel algorithm on GPUs," *IEEE Trans. Comput.*, vol. 62, no. 10, pp. 1906–1916, Oct. 2013.

[141] X. Yu and M. Becchi, "Exploring different automata representations for efficient regular expression matching on GPUs," *ACM SIGPLAN Not.*, vol. 48, no. 8, pp. 287–288, 2013.

[142] D. P. Scarpazza, O. Villa, and F. Petrini, "Peak-performance DFA-based string matching on the cell processor," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, Long Beach, CA, USA, 2007, pp. 1–8.

[143] J. Jiang, X. Wang, K. He, and B. Liu, "Parallel architecture for high throughput DFA-based deep packet inspection," in *Proc. IEEE Int. Conf. Commun. (ICC)*, Cape Town, South Africa, 2010, pp. 1–5.

[144] J. Holub and S. Štekr, "On parallel implementations of deterministic finite automata," in *Implementation and Application of Automata*. Heidelberg, Germany: Springer, 2009, pp. 54–64.

[145] D. Luchaup, R. Smith, C. Estan, and S. Jha, "Multi-byte regular expression matching with speculation," in *Recent Advances in Intrusion Detection*. Heidelberg, Germany: Springer, 2009, pp. 284–303.

[146] D. Luchaup, R. Smith, C. Estan, and S. Jha, "Speculative parallel pattern matching," *IEEE Trans. Inf. Forensics Security*, vol. 6, no. 2, pp. 438–451, Jun. 2011.

[147] Y. Ko, M. Jung, Y.-S. Han, and B. Burgstaller, "A speculative parallel DFA membership test for multicore, SIMD and cloud computing environments," *Int. J. Parallel Program.*, vol. 42, no. 3, pp. 456–489, 2014.

[148] Y.-H. E. Yang and V. K. Prasanna, "Optimizing regular expression matching with SR-NFA on multi-core systems," in *Proc. Int. Conf. Parallel Architect. Compilation Techn. (PACT)*, Galveston, TX, USA, 2011, pp. 424–433.

[149] F. Yu, R. H. Katz, and T. V. Lakshman, "Gigabit rate packet pattern-matching using TCAM," in *Proc. 12th IEEE Int. Conf. Netw. Protocols (ICNP)*, Berlin, Germany, 2004, pp. 174–183.

[150] J.-S. Sung, S.-M. Kang, Y. Lee, T.-G. Kwon, and B.-T. Kim, "A multi-gigabit rate deep packet inspection algorithm using TCAM," in *Proc. Glob. Telecommun. Conf. (GLOBECOM)*, vol. 1. St Louis, MO, USA, 2005, p. 5.

[151] M. Alicherry, M. Muthuprasanna, and V. Kumar, "High speed pattern matching for network IDS/IPS," in *Proc. 14th IEEE Int. Conf. Netw. Protocols (ICNP)*, Santa Barbara, CA, USA, 2006, pp. 187–196.

[152] K. Peng, S. Tang, M. Chen, and Q. Dong, "Chain-based DFA deflation for fast and scalable regular expression matching using TCAM," in *Proc. IEEE 7th ACM/IEEE Symp. Architect. Netw. Commun. Syst. (ANCS)*, Brooklyn, NY, USA, 2011, pp. 24–35.

[153] C. R. Meiners, J. Patel, E. Norige, E. Torng, and A. X. Liu, "Fast regular expression matching using small TCAMs for network intrusion detection and prevention systems," in *Proc. 19th USENIX Conf. Security*, Berkeley, CA, USA, 2010, pp. 1–16.

[154] C. R. Meiners, J. Patel, E. Norige, A. X. Liu, and E. Torng, "Fast regular expression matching using small TCAM," *IEEE/ACM Trans. Netw. (TON)*, vol. 22, no. 1, pp. 94–109, Feb. 2014.

[155] J. V. Lunteren and A. Guanella, "Hardware-accelerated regular expression matching at multiple tens of GB/s," in *Proc. IEEE INFOCOM*, Orlando, FL, USA, 2012, pp. 1737–1745.

[156] H. Nakahara, T. Sasao, and M. Matsuura, "A regular expression matching circuit based on a decomposed automaton," in *Reconfigurable Computing: Architectures, Tools and Applications*. Heidelberg, Germany: Springer, 2011, pp. 16–28.

[157] J. V. Lunteren *et al.*, "Designing a programmable wire-speed regular-expression matching accelerator," in *Proc. 45th Annu. IEEE/ACM Int. Symp. Microarchitect. (MICRO)*, Vancouver, BC, Canada, 2012, pp. 461–472.

[158] K. Atasu, F. Doerfler, J. V. Lunteren, and C. Hagleitner, "Hardware-accelerated regular expression matching with overlap handling on IBM PowerEn processor," in *Proc. IEEE 27th Int. Symp. Parallel Distrib. Process. (IPDPS)*, Boston, MA, USA, 2013, pp. 1254–1265.

[159] A. X. Liu, E. Norige, and S. Kumar, "A few bits are enough-ASIC friendly regular expression matching for high speed network security systems," in *Proc. 21st IEEE Int. Conf. Netw. Protocols (ICNP)*, Göttingen, Germany, 2013, pp. 1–10.

[160] P. Dlugosch, D. Brown, P. Glendenning, M. Leventhal, and H. Noyes, "An efficient and scalable semiconductor architecture for parallel automata processing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 12, pp. 3088–3098, Dec. 2014.

[161] *New Hogwash Web Site*. Accessed on Feb. 20, 2016. [Online]. Available: http://hogwash.sourceforge.net/oldindex.html

[162] *Stratix ii Device Handbook: Volume 1*. Accessed on Feb. 20, 2016. [Online]. Available: http://www.altera.com/literature/hb/stx2/stx2_sii5v1.pdf

[163] *Cuda C Programming Guide-v7.5*. (2015). [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide/#axzz3lG7i9ZOS

[164] *Cisco Intusion Prevention System (Cisco Ips)*. Accessed on Feb. 20, 2016. [Online]. Available: http://www.cisco.com/en/US/products/ps6634/products_ios_protocol_group_home.html

[165] M. Becchi, C. Wiseman, and P. Crowley, "Evaluating regular expression matching engines on network and general purpose processors," in *Proc. 5th ACM/IEEE Symp. Architect. Netw. Commun. Syst.*, Princeton, NJ, USA, 2009, pp. 30–39.

[166] *Defcon*. (2013). [Online]. Available: http://cctf.shmoo.com.

[167] M. Becchi, M. Franklin, and P. Crowley, "A workload for evaluating deep packet inspection architectures," in *Proc. IEEE Int. Symp. Proc. Workload Characterization (IISWC)*, Seattle, WA, USA, 2008, pp. 79–89.

[168] M. Najam, U. Younis, and R. ur Rasool, "Speculative parallel pattern matching using stride-k DFA for deep packet inspection," *J. Netw. Comput. Appl.*, vol. 54, pp. 78–87, 2015.

[169] R. Sommer, V. Paxson, and N. Weaver, "An architecture for exploiting multi-core processors to parallelize network intrusion prevention," *Concurrency Comput. Pract. Exp.*, vol. 21, no. 10, pp. 1255–1279, 2009.

[170] J. Jung, V. Paxson, A. W. Berger, and H. Balakrishnan, "Fast portscan detection using sequential hypothesis testing," in *Proc. IEEE Symp. Security Privacy*, Berkeley, CA, USA, 2004, pp. 211–225.

[171] D. R. Ellis, J. G. Aiken, K. S. Attwood, and S. D. Tenaglia, "A behavioral approach to worm detection," in *Proc. ACM Workshop Rapid malcode*, Washington, DC, USA, 2004, pp. 43–53.

[172] Y. Zhang and V. Paxson, "Detecting stepping stones," in *Proc. USENIX Security Symp.*, vol. 9, 2000, p. 13.

[173] V. Paxson *et al.*, "Rethinking hardware support for network analysis and intrusion prevention," in *Proc. HotSec*, 2006, Art. no. 11.

[174] S. Singh, C. Estan, G. Varghese, and S. Savage, "Automated worm fingerprinting," in *Proc. OSDI*, vol. 6. San Francisco, CA, USA, 2004, Art. no. 4.

[175] M. Gokhale *et al.*, "Granidt: Towards gigabit rate network intrusion detection technology," in *Field-Programmable Logic and Applications: Reconfigurable Computing Is Going Mainstream*. Heidelberg, Germany: Springer, 2002, pp. 404–413.

[176] *Clamav*. (2015). [Online]. Available: http://www.clamav.net/index.html

[177] S. Yun, "An efficient TCAM-based implementation of multipattern matching using covered state encoding," *IEEE Trans. Comput.*, vol. 61, no. 2, pp. 213–221, Feb. 2012.

[178] K. Zheng, Z. Cai, X. Zhang, Z. Wang, and B. Yang, "Algorithms to speedup pattern matching for network intrusion detection systems," *Comput. Commun.*, vol. 62, pp. 47–58, May 2015.

[179] K. Huang *et al.*, "Scalable TCAM-based regular expression matching with compressed finite automata," in *Proc. 9th ACM/IEEE Symp. Architect. Netw. Commun. Syst.*, San Jose, CA, USA, 2013, pp. 83–93.

[180] *Dansguardian True Web Content Filtering for All*. (2015). [Online]. Available: http://dansguardian.org/

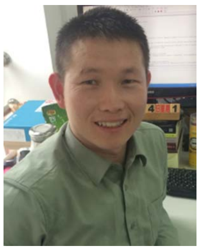
[181] *Graphviz - Graph Visualization Software*. (2011). [Online]. Available: http://www.graphviz.org/content/new-gvedit-released

[182] S. Chen and R. Lu, "A regular expression matching engine with hybrid memories," *Comput. Stand. Interfaces*, vol. 36, no. 5, pp. 880–888, 2014.

[183] J. G. Kemeny and J. L. Snell, *Finite Markov Chains*, vol. 356. Princeton, NJ, USA: Van Nostrand, 1960.

[184] B. Khailany *et al.*, "Imagine: Media processing with streams," *IEEE Micro*, vol. 21, no. 2, pp. 35–46, Mar./Apr. 2001.

[185] M. Becchi, A. Bremler-Barr, D. Hay, O. Kochba, and Y. Koral, "Accelerating regular expression matching over compressed http," in *Proc. (INFOCOM)*, Apr./May 2015, pp. 540–548.



Chengcheng Xu received the B.S. degree from the Nanjing University of Aeronautics and Astronautics, China, in 2011, and the M.S. degree from the National University of Defense Technology, China, in 2013, where he is currently pursuing the Ph.D. degree. His interests are in network security and deep packet inspection.

Shuhui Chen received the Ph.D. degree from the National University of Defense Technology, China, in 2007, where he is currently a Professor. His research interests include network protocol and network security.

Jinshu Su received the B.S. degree in mathematics from Nankai University, Tianjin, China, in 1985, and the M.S. and Ph.D. degrees in computer science from the National University of Defense Technology, Changsha, China, in 1988 and 2000, respectively.

He is a Professor with the School of Computer Science, National University of Defense Technology. He currently leads the Distributed Computing and High Performance Router Laboratory and the Computer Networks and Information Security Laboratory, which are both key laboratories of National 211 and 985 projects, China. He also leads the High Performance Computer Networks Laboratory, which is a key laboratory of Hunan Province, China. His research interests include Internet architecture, Internet routing, security, and wireless networks.

S. M. Yiu received the Ph.D. degree in computer science from the Department of Computer Science, University of Hong Kong, in 1996, where he is currently an Associate Professor. His research interests include information security, cryptography, and bioinformatics.

Lucas C. K. Hui received the B.Sc. and M.Phil. degrees in computer science from the University of Hong Kong, and the M.Sc. and Ph.D. degrees in computer science from the University of California, Davis. He is the Founder and the Honorary Director of the Center for Information Security and Cryptography, and concurrently an Associate Professor with the Department of Computer Science, University of Hong Kong. His research interests include information security, computer crime, cryptographic systems, and electronic commerce security. He is a member of the HKIE.