Advances in Deep Packet Inspection Technology: Postprint

Authors: Huang Kun, Xie Gaogang

Date: 2017-03-10T00:00:00+00:00

Abstract

Deep packet inspection is one of the key technologies in packet processing, which employs pattern matching algorithms to match the content of each packet against a set of predefined patterns. With the rapid growth of network bandwidth and the increasing number of pattern rules, researchers have proposed hardware-based pattern matching algorithms, i.e., designing and implementing pattern matching algorithms using specialized embedded hardware such as FPGA, ASIC, and NP to improve the matching throughput of DPI. However, these hardware-based pattern matching algorithms face high-performance challenges, namely how to satisfy the time and space requirements of line-rate packet content filtering. This paper surveys the research progress of hardware-based string matching algorithms and regular expression matching algorithms from the perspectives of time and space, and prospects future DPI technology research.

Full Text

Abstract

Deep Packet Inspection (DPI) represents a critical technology in packet processing that employs pattern matching algorithms to compare packet contents against a set of predefined signatures. With the rapid growth of network bandwidth and the increasing number of signature rules, researchers have proposed hardware-based pattern matching algorithms implemented on specialized embedded hardware such as FPGAs, ASICs, and network processors to enhance DPI matching throughput. However, these hardware-based approaches face significant performance challenges in meeting the stringent time and space requirements for line-rate packet content filtering. This paper surveys recent advances in hardware-based string matching and regular expression matching algorithms from temporal and spatial perspectives, and outlines future research directions in DPI technology.

Keywords: network security, deep packet inspection, pattern matching, string matching, regular expression

1 Introduction

The rapid development and widespread adoption of network technologies have enabled the Internet to carry an ever-growing variety of applications, with user dependency on the Internet increasing daily. In recent years, novel attacks such as network worms, botnets, and computer viruses have emerged incessantly, compromising vast numbers of computer systems, abusing network resources, and threatening Internet infrastructure, resulting in substantial economic losses and severe social impacts [1-2]. Consequently, the rapid detection and prevention of such attacks have become pressing issues in network security research.

Network Intrusion Detection/Prevention Systems (NIDS/NIPS) constitute the primary means of network security defense, operating by monitoring network traffic in real-time to inspect and block network attacks [1]. NIDS/NIPS have been widely deployed across network components ranging from end-user computers and edge routers to core routers. As network attacks grow increasingly sophisticated and security vulnerabilities continue to be discovered, the global NIDS/NIPS market is projected to grow from $932 million in 2007 to $2.1 billion within five years [3]. Deep Packet Inspection (DPI) forms the core of NIDS/NIPS, examining not only packet headers but also packet payloads (i.e., packet contents). DPI primarily employs pattern matching algorithms that compare each packet' s content against a set of predefined signature rules [4-5].

DPI utilizes a collection of rules to describe attack signatures, where each rule specifies at minimum the packet type, signature string, search start position, and response action upon matching. For instance, in the Snort rule set [6], the rule `{alert icmp (msg:"DDOS TFN Probe"; content:"1234";)}` describes a distributed denial-of-service TFN probe attack, generating an alert message "DDOS TFN Probe" when an Internet Control Message Protocol (ICMP) packet contains the signature string "1234" . As a packet content filtering technology, DPI finds application not only in NIDS/NIPS but also in application-layer packet classification, peer-to-peer (P2P) traffic identification, and context-aware traffic billing [7-8].

Pattern matching represents a classic problem in computer science with broad applications in information retrieval, pattern recognition, and network security. Pattern matching algorithms can be categorized into string matching and regular expression matching algorithms. String matching employs string-based languages to describe simple signatures, while regular expression matching uses regular expression languages to describe complex signatures. Based on matching methodology, these algorithms further divide into single-pattern and multi-pattern approaches. Single-pattern algorithms scan content once to match a single signature, exemplified by the Knuth-Morris-Pratt [9] and Boyer-Moore [10] algorithms. When a rule set contains s signatures, single-pattern algorithms re-

quire s repetitive matching operations, resulting in low efficiency. Multi-pattern algorithms employ Deterministic Finite Automata (DFA) to represent a set of signatures, enabling a single content scan to match multiple signatures simultaneously, as seen in the Aho-Corasick [11], Commentz-Walter [12], and Aho-Corasick-Boyer-Moore hybrid algorithms [13].

Since 1975, nearly three decades of DPI research have focused primarily on software-based pattern matching algorithms implemented on single-core CPU platforms to improve matching rates. However, with explosive growth in network bandwidth, traffic volume, and signature rules, software-based approaches suffer from low matching throughput, failing to meet the high-performance demands of 10-40 Gbps line-rate packet processing. In recent years, researchers have proposed various hardware-based pattern matching algorithms [14-18] implemented on specialized embedded hardware such as Field-Programmable Gate Arrays (FPGAs) and Application-Specific Integrated Circuits (ASICs) to enhance DPI throughput. For example, FPGA-based implementations can achieve 10 Gbps throughput while supporting dynamic rule updates, albeit with lengthy recompilation and reconfiguration times, whereas ASIC-based approaches can reach 20 Gbps but lack support for dynamic updates. These hardware-based algorithms face challenges from limited embedded memory, as DFA storage requirements often exceed on-chip high-speed memory capacity, necessitating frequent off-chip memory accesses that degrade performance. For instance, the Xilinx Vertex-5 FPGA [19] provides only approximately 10 Mb of on-chip SRAM, while the DFA for the current Snort rule set requires about 100 MB, making on-chip storage infeasible and preventing line-rate processing. Advances in Ternary Content Addressable Memory (TCAM) and multi-core processor technologies present new opportunities and challenges for DPI performance enhancement. TCAM can perform content lookup in a single clock cycle, providing high-speed deterministic search to accelerate DPI matching, while multi-core processors integrate multiple CPU cores on a single chip, offering powerful parallel computing capabilities. Consequently, designing and implementing high-performance hardware-based pattern matching algorithms for line-rate DPI has become a hot research topic attracting growing attention.

This paper surveys advances in DPI technology, focusing on hardware-based pattern matching algorithm design and implementation. Section 2 discusses several hardware-based string and regular expression matching algorithms. Finally, Section 5 concludes and outlines key issues for future DPI research.

2 High-Performance Challenges in DPI Technology

DPI constitutes a computationally intensive operation deployed on the critical data path of high-speed routers, requiring inspection of massive packet streams against thousands of rules. In recent years, Internet backbone link bandwidth has increased from 2.5 Gbps to 10-40 Gbps, with Ethernet interfaces advancing from 10GbE to 100GbE. New high-quality applications such as streaming media (PPLive, YouTube) and content distribution platforms (Facebook, Twitter)

generate enormous traffic volumes. To accommodate high-speed massive packet processing, meet line-rate temporal and spatial requirements, and enhance scalability, DPI technology faces significant performance challenges [20].

First, as network attacks and application behaviors grow increasingly complex, signature rule sets expand continuously with more sophisticated descriptions, leading to escalating storage overhead. For example, the Snort rule set grew from 3,166 rules in 2003 to 15,047 in 2009, with its DFA requiring over 75 MB of storage. Since embedded hardware like FPGAs and ASICs provides only about 10 Mb of on-chip high-speed memory, the complete DFA cannot reside on-chip and must be stored in slower off-chip memory, limiting DPI throughput. Therefore, hardware-based DPI urgently requires memory-efficient pattern matching algorithms that store and lookup entire DFAs in on-chip memory to satisfy scalability demands while improving performance.

Second, to achieve future 100 Gbps line-rate processing, hardware-based DPI must leverage on-chip high-speed memory and parallel computing capabilities to increase matching throughput. Embedded hardware typically employs hierarchical memory architectures comprising on-chip high-speed and off-chip low-speed memory. On-chip memory supports fast lookup (e.g., 1-2 ns access time for on-chip SRAM) but offers limited capacity, while off-chip memory provides larger capacity but slower access (e.g., 60 ns for off-chip DRAM). Hardware-oriented DPI must minimize off-chip memory accesses to achieve line-rate performance. Although specialized embedded hardware supports parallel processing, it suffers from complex programming, limited flexibility, and high upgrade costs. Multi-core processors such as Intel Xeon CPUs, Cavium OCTEON, and Nvidia GPGPUs offer powerful parallel computing with flexible programmability, presenting new opportunities and challenges. Line-rate DPI must exploit multi-core parallelism to design and implement spatiotemporally efficient pattern matching algorithms that accelerate performance.

In summary, DPI scalability demands manifest in two dimensions: storage overhead and matching throughput [21], requiring reduced storage space while increasing matching rates to achieve high-performance DPI. Current DPI research primarily focuses on designing hardware-based pattern matching algorithms from temporal and spatial perspectives to meet these demands. This paper examines advances in hardware-based string and regular expression matching algorithms through a spatiotemporal lens and identifies key future research issues.

3 Hardware-Based String Matching Algorithms

Hardware-based string matching algorithms can be classified into memory-efficient, multi-character, and parallel string matching algorithms. As shown in Table 1 , memory-efficient algorithms compress DFA storage by eliminating redundant transition edges, exemplified by B-FSM-based [22] and CDFA-based [23] Aho-Corasick implementations. Multi-character algorithms improve

throughput by constructing DFAs that process multiple characters per cycle, including JACK-NFA-based [24], TDP-DFA-based [25], compressed DFA [26], and variable-stride DFA [27] variants of Aho-Corasick. Parallel algorithms accelerate DFA matching through parallelization, reducing storage requirements while increasing throughput, such as bit-split Aho-Corasick [28], head-body partitioned Aho-Corasick [29], and parallel Bloom filter-based [30] approaches.

3.1 Aho-Corasick Algorithm

The Aho-Corasick algorithm [11] represents a classic string matching approach that uses a DFA to encode a set of pattern strings, known as the original DFA. The construction process involves three steps: building a trie rooted at initial state 0 from the pattern set; extracting all unique characters to form an alphabet; and constructing transition (Goto), failure, and output functions for each state from the root to leaf nodes. The transition function defines state migration when an input character matches, the failure function specifies migration on mismatch, and the output function reports matched strings upon reaching accepting states.

Figure 1 [Figure 1: see original paper] illustrates the original DFA for the string set `{he, she, his, hers}`, with initial state 0 and accepting states 2, 5, 7, and 9. Solid lines denote basic transitions from a source state at depth d to a destination state at depth $d+1$, while dashed lines represent cross transitions to states at depth $d' \quad d$. Cross transitions to depth 0 are failure transitions, and those to depth 1 are restart transitions. The state transition table uses source states as row headers and destination states as columns, with matching characters as column headers (failure transitions are omitted for clarity).

The matching process begins with current state 0, migrating to the next state upon each input character and updating the current state sequentially. When reaching an accepting state, all matched strings are output. For example, processing the string `{sohershe}` starting from state 0 yields the state sequence `0 → → → → → → → →` and outputs the matched set `{he, she, his, hers}`.

Tuck et al. [31] employed bitmap compression and path compression to reduce DFA storage requirements and improve worst-case performance. Since the ASCII alphabet contains 256 characters, each DFA state originally maintains 256 pointers, causing substantial memory overhead. As shown in Figure 2 [Figure 2: see original paper], the bitmap-compressed DFA uses a single state pointer (Next Ptr), a node bitmap (Node Bitmap), a failure pointer (Failure Ptr), and a rule pointer (Rule Ptr). The state pointer points to the head of the next-state queue, the node bitmap indicates whether each ASCII character' s next state is the initial state (0 for initial, 1 otherwise), the failure pointer points to the initial state, and the rule pointer references the matched rule queue.

During matching, when reading character r with its bitmap bit set to 1, the algorithm counts preceding 1-bits in the Node Bitmap to compute the next state' s address and checks its rule pointer for matches. When reading character i with its bit set to 1, the next state address points to the initial state via the

failure pointer.

3.2 Memory-Efficient String Matching Algorithms

DFA storage space comprises state transition edges. Memory-efficient algorithms reduce storage requirements by eliminating redundant transitions. This section covers B-FSM-based and CDFA-based Aho-Corasick implementations.

3.2.1 B-FSM-Based Aho-Corasick Algorithm Van Lunteren et al. [22] proposed a B-FSM (BART-based Finite State Machine) approach that compresses transitions using priorities and wildcard states, employing the BART (Balanced Routing Table) algorithm to locate the highest-priority matching transition. Figure 3 [Figure 3: see original paper] shows the B-FSM for `{he, she, his, hers}`, with transition diagrams on the left and prioritized edges on the right. Each transition contains four fields: source state, matching character, destination state, and priority. For example, the transition `{1 → 2: e, priority 2}` indicates migration from state 1 to state 2 on character e with priority 2, while `{* → 1: h, priority 1}` denotes migration from any state to state 1 on character h with priority 1, and `{* → 0:  , priority 0}` represents migration from any state to state 0 on non-alphabet characters with priority 0.

The B-FSM matching process selects the highest-priority transition matching the current state and input character, searching from priority 2 down to 0. Processing `{sohershe}` begins at state 0: reading s matches `{* → 3: s, priority 1}`; o matches `{* → 0:  , priority 0}`; the second h matches `{* → 1: h, priority 1}`; e matches `{1 → 2: e, priority 2}`, outputting `{he}`; r matches `{2 → 8: r, priority 2}`; s matches `{8 → 9: s, priority 2}`, outputting `{hers}`; the third h matches `{9 → 4: h, priority 2}`; and the final e matches `{4 → 5: e, priority 2}`, outputting `{she, he}`. The complete state sequence `0 → → → → → → → →` produces matches `{he, hers, she, he}`.

3.2.2 CDFA-Based Aho-Corasick Algorithm Song et al. [23] proposed a CDFA (Cached DFA) approach that records additional states in a cache to extend the DFA, reducing redundant cross transitions while enabling efficient transition storage and lookup. Figure 4 [Figure 4: see original paper] shows the CDFA for `{he, she, his, hers}`. Compared to B-FSM, CDFA eliminates some cross transitions (those to depth-2 states) while retaining others, using a cache to store one additional current state.

The CDFA matching process initializes `current_state` and `cache_state` to 0. Upon each input character, it looks up `next_state` for both `current_state` and `cache_state`. If `next_state` exists for `current_state`, it updates `current_state` to `next_state` and `cache_state` to the previous `current_state`. If `next_state` exists only for `cache_state`, it sets `current_state` to that `next_state`. If neither exists, it sets `cache_state` to `next_state` from the initial state. Processing `{shishe}`: reading s sets both

states to 3; h finds `next_state` 4 for `current_state` and 0 for `cache_state`, updating `current_state` to 4 and `cache_state` to 0; i finds no transition for `current_state` but `next_state` 7 for `cache_state`, setting `current_state` to 7 and `cache_state` to 3, outputting `{his}`; s finds `next_state` 3 for `current_state`, updating both states to 3; h finds `next_state` 4 for `current_state` and 1 for `cache_state`; and e finds `next_state` 2 for `current_state` and 5 for `cache_state`, outputting `{she, he}`.

3.3 Multi-Character String Matching Algorithms

To improve throughput, researchers have developed multi-character algorithms that process multiple characters per cycle, including fixed-stride and variable-stride approaches.

3.3.1 Fixed-Stride Multi-Character Aho-Corasick Algorithm Dharmapurikar et al. [24] proposed JACK-NFA (Jump-ahead Aho-Corasick NFA), which builds an NFA that jumps ahead by a fixed stride and uses Bloom filters for transition storage and lookup. Figure 5 [Figure 5: see original paper] shows the JACK-NFA for `{abcd, cde, bade, bc}` with stride 2. Transitions are grouped by match length and represented using Bloom filters, with each table entry containing `<source state, match character>`, destination state, extra state, and matched strings.

The matching process initializes the current state to 0 and employs two DFAs (DFA1 and DFA2) processing offset-by-one character streams in parallel. Each character lookup in the transition table' s Bloom filter migrates both DFAs to their next states, outputting matches when reaching accepting states. Processing `{abadeabcdea}`: DFA1 reads `{abadeabcdea}` while DFA2 reads `{badeabcdea}`; DFA1' s state sequence `0 → → → → → →` outputs `{bc}`, while DFA2' s sequence `0 → → → → → →` outputs `{bade, abcd, cde}`.

Lu et al. [25] proposed TDP-DFA (Transition-Distributed Parallel DFA), which similarly jumps fixed strides but uses BCAM (Binary Content Addressable Memory) for parallel transition lookup. Figure 6 [Figure 6: see original paper] shows the TDP-DFA for `{abcd, cde, bade, bc}` with stride 2, using `<source state, match character>` as search keys in BCAM. Each entry contains destination state and match flag, with matched strings stored off-chip. The matching process parallels JACK-NFA, with DFA1 and DFA2 executing concurrently.

Alicherry et al. [26] proposed a compressed DFA that processes multiple characters, supporting rollback and longest-match operations with TCAM-based transition lookup. Figure 7 [Figure 7: see original paper] shows the compressed DFA for `{abcd, cde, bade, bc}` with stride 2. Each entry contains `<source state, match character>`, destination state, stride offset, and matched strings, prioritized from basic to restart to split to failure transitions. Processing `{abadeabcdea}` yields the state sequence `0 → {6} → {1} → {1} → {8} → {5} → {4}` with outputs `{bade, abcd, bc, cde}`.

3.3.2 Variable-Stride Multi-Character Aho-Corasick Algorithm Hua et al. [27] proposed a variable-stride DFA using winnowing algorithms to split patterns and input into variable-size substrings, constructing a DFA with hash tables for core transitions and TCAM for short transitions. Figure 8 [Figure 8: see original paper] shows the variable-stride DFA for `{power, identical, authenticate, enter, set}`. The winnowing algorithm splits patterns into head, core, and tail substrings, with core transitions (solid lines) looked up via hash tables and short transitions (dashed lines) via TCAM to reach accepting states (dashed nodes).

3.4 Parallel String Matching Algorithms

To achieve line-rate DPI, parallel algorithms exploit parallelism to accelerate DFA matching and compress storage requirements.

3.4.1 Bit-Split Aho-Corasick Algorithm Tan et al. [28] proposed bit-split Aho-Corasick for embedded hardware, decomposing the original DFA into parallel micro-DFAs that process specific bit substrings of each input character. Figure 9 [Figure 9: see original paper] shows the bit-split DFA for `{he, she, his, hers}`. The original alphabet is split into micro-alphabets (e.g., `01BΣ` and `23BΣ`), where each 8-bit character is divided into four 2-bit substrings. Micro-alphabet `01BΣ` includes tuples of bits 0-1 and corresponding character subsets (e.g., tuple 01 maps to `{e,i}`), while `23BΣ` includes bits 2-3.

The matching process distributes each character' s b-bit substrings to corresponding micro-DFAs, which execute in parallel from initial micro-state 0' . The intersection of resulting micro-states yields the original matched state and patterns. Processing `{rhe}`: `01B DFA` reads substrings `{10,00,01}` and executes `0' → → →`, while `23B DFA` reads `{10,10,01}` and executes `0' → → →`. The intersection of states `3'` and `6'` outputs matched state `{2}` and pattern `{he}`.

3.4.2 Head-Body Partitioned Aho-Corasick Algorithm Yang et al. [29] proposed head-body partitioned Aho-Corasick, splitting the DFA into one head DFA and multiple parallel tail NFAs, implemented on multi-core processors using multithreading. Figure 10 [Figure 10: see original paper] shows the partitioned DFA for `{he, she, his, hers}` split at depth 2. States 2, 6, and 4 are trigger states that launch tail NFA threads when reached. On multi-core platforms, the head DFA runs as the main thread while tail NFAs execute as parallel child threads.

The matching process migrates the head DFA on each character, launching tail threads at trigger states that execute in parallel. Processing `{sohershe}`: the head DFA sequence `0 → → → → → → → →` triggers tail threads at states 2 and 4, which execute `2 → 5 →` and `4 → →` respectively, outputting `{he, she, his, hers}`.

3.4.3 Parallel Bloom Filter-Based Aho-Corasick Algorithm Dharmapurikar et al. [30] proposed a parallel Bloom filter approach that partitions patterns by length into subsets, each represented by a Bloom filter [32] for parallel lookup. Figure 11 [Figure 11: see original paper] shows the architecture with parallel Bloom filters for equal-length pattern subsets. The algorithm extracts substrings of increasing length from input and queries filters in parallel, outputting successful matches. Due to Bloom filter false positives [32], the algorithm may produce false matches requiring off-chip hash table verification, suffering from high memory bandwidth demands, cost, and limited scalability.

4 Hardware-Based Regular Expression Matching Algorithms

As intelligent attacks such as evasion [33] and polymorphic/metamorphic attacks [34-35] proliferate, strings inadequately describe complex signatures, reducing detection rates. Regular expressions' rich expressiveness [36] has led NIDS/NIPS to adopt them for signature description.

Table 2 classifies hardware-based regular expression matching into transition-compression and state-compression algorithms. Transition-compression algorithms like D2FA [37] and CD2FA [38] reduce redundant transitions, while state-compression algorithms such as state-merged DFA [39] and Extended Finite Automaton (XFA) [40-41] eliminate redundant states.

4.1 Original Regular Expression Matching Algorithm

Original DFA construction involves pattern compilation and subset construction [37]. Similar to Aho-Corasick, compilation builds a directed trie from initial to accepting states, while subset construction creates deterministic transitions by converting NFA to DFA. Regular expressions feature special symbols: `+` (one or more), `*` (zero or more), `·` (concatenation), `.` (wildcard), `?` (optional), `{}` (repetition), `[]` (character class), and `[^]` (negated class). Figure 12 [Figure 12: see original paper] shows the original DFA for `{ab*c, de*}` with transition table column `*` representing characters other than `{a,b,c,d,e}` for failure transitions.

The matching process begins at state 0, migrating on each character and outputting matches at accepting states. Processing `{abadcade}` yields the sequence `0 → → → → → → → →` and outputs `{ab*c, de*}`. DFA size depends on state count and transitions per state. For regular expressions, storage explosion stems from: (1) numerous redundant inter-state and intra-state transitions, and (2) extensive extra states required to track partial matches for semantic symbols like `*`, `{}`, and `[]`, prompting research into state and transition compression.

4.2 Transition-Compression Regular Expression Matching Algorithms

4.2.1 D2FA-Based Algorithm Kumar et al. [37] proposed D2FA, replacing identical transitions across states with default transitions while preserving

distinct transitions to reduce storage. Figure 13 [Figure 13: see original paper] shows D2FA for {ab*c, de*}, where dashed lines denote default transitions and solid lines denote normal transitions. Processing character a at state 5 follows default transitions 5 → 4 → 0 before taking normal transition 0 → 1. While D2FA significantly reduces storage, it increases transition counts: {abadcade} requires 8 transitions in the original DFA but 10 in D2FA, reducing matching throughput.

4.2.2 CD2FA-Based Algorithm Kumar et al. [38] proposed CD2FA, enhancing D2FA with content identifiers at each state to accelerate matching at the cost of additional storage. Figure 14 [Figure 14: see original paper] shows CD2FA for {ab*c, de*}, where each state maintains identifiers for characters along default paths. State 5' s identifier 0_,e indicates null at state 5, effective character e at state 4, and root node 0. States 0 and 2 contain only self-identifiers.

CD2FA matching uses these identifiers to jump along default paths when normal transitions fail, reducing transition counts. Processing {abadcade} executes 0 → → → → → → → →, matching the original DFA's transition count while significantly reducing storage.

4.3 State-Compression Regular Expression Matching Algorithms

4.3.1 State-Merged DFA Algorithm Becchi et al. [39] proposed state merging using transition labeling to fuse non-equivalent states, reducing storage while preserving worst-case performance. Figure 15 [Figure 15: see original paper] shows the state-merged DFA for {(abc)+, def*g+}. Merged state 3-4 has labeled transitions 0· and 1· that set marker bits, and 0/ and 1/ that check markers before migrating. Processing {abcdefg} yields the sequence 0 → → → → → → → → and outputs {(abc)+, def*g+}.

4.3.2 Extended Finite Automaton (XFA) Algorithm Smith et al. [40-41] proposed XFA, adding auxiliary variables and simple instructions to states to avoid state explosion. DFA explosion occurs when combining individual DFAs, as non-ambiguous states from one interact with ambiguous states from others, creating numerous extra states to track partial matches. XFA replaces these extra states with variables recording match progress and uses instructions (Set, Reset, Compare) to verify completion.

Figure 16 [Figure 16: see original paper] shows XFA for {abc*de, f*gh}. The individual XFA for abc*de adds set 1bit=1 at state 3 to record abc match and if(1bit==1) output at state 5. The combined XFA uses variables 1bit and 2bit at states 3' and 7' , and comparison instructions at states 5' and 8' , requiring only 9 states and 2 auxiliary bits. Processing {abcfhde} executes 0' → → → → → → → and outputs {abc*de}.

5 Conclusion and Outlook

DPI technology faces high-performance challenges in meeting storage and throughput demands for high-speed packet processing while enhancing pattern matching scalability. This paper surveyed hardware-based string and regular expression matching algorithms from spatiotemporal perspectives, focusing on memory-efficient, multi-character, and parallel string matching algorithms, as well as transition- and state-compression techniques for regular expressions. Given modern embedded hardware' s limited memory yet powerful parallel computing capabilities, hardware-based pattern matching research aims to compress DFA storage to enable on-chip storage and lookup, achieving high-throughput matching.

As 100 Gbps networks become prevalent, future DPI faces scalability challenges in processing rates, rule set size, power consumption, and cost. Advances in high-speed memory and multi-core processors present new opportunities. Key future research directions include:

1. **Multi-core processor-based parallel DPI**: Multi-core CPUs and GPGPUs integrate hundreds of cores, offering massive parallelism. Research must explore fine-grained parallelization of pattern matching algorithms [42] to achieve data, task, and pipeline parallelism across cores while developing memory-efficient algorithms that store entire DFAs in per-core caches to accelerate throughput.
2. **TCAM-based efficient DPI**: As TCAMs become faster, larger, and more power-efficient, TCAM-based DPI can be deployed on national backbone links for high-speed, high-capacity packet classification. Since TCAM power consumption scales with storage size, research must focus on compressing DFA storage [43] by reducing both transitions and states to minimize power and enhance performance.
3. **High-performance DPI for network virtualization**: Next-generation Internet projects like GENI and FIND employ network virtualization to run multiple isolated virtual networks on shared infrastructure. Programmable network devices must support multiple parallel pattern matching algorithms within limited compute and storage resources, ensuring each algorithm' s storage and throughput requirements to improve scalability for virtualized network functions.

References

[1] S. Staniford, V. Paxson and N. Weaver, How to own the Internet in your spare time. In: Proceedings of USENIX Security Symposium, 2002.

[2] S. Singh, C. Estan and G. Varghese and S. Savage, Automated worm fingerprinting. In: Proceedings of OSDI, 2004.

[3] Frost and Sullivan. World intrusion detection and prevention systems markets. 2007.

[4] M. Roesch, Snort -lightweight intrusion detection for networks. In: Proceedings of LISA, 1999.

[5] V. Paxon, Bro: a system for detecting network intruders in real-time. Computer Networks, 1999, 31(23-24): 2435-2463.

[6] Snort-the de facto standard for intrusion detection/prevention. http://www.snort.org, 2009.

[7] J. Levandoski, E. Sommer and M. Strait, Application layer packet classifier for Linux. http://l7-filter.sourceforge.net/, 2008.

[8] S. Sen, O. Spatscheck and D. Wang, Accurate, scalable in-network identification of P2P traffic using application signatures. In: Proceedings of WWW, 2004.

[9] D. E. Knuth, J. H. Morris and V. R. Pratt, Fast pattern matching in strings. SIAM Journal on Computing, 1977, 6(2): 323-350.

[10] R. S. Boyer and J. S. Moore, A fast string searching algorithm. Communications of the ACM, 1977, 20(10): 762-772.

[11] A. V. Aho and M. J. Corasick, Efficient string matching: an aid to bibliographic search. Communications of the ACM, 1975, 18(6): 333-340.

[12] B. Commentz-Walter, A string matching algorithm fast on the average. In: Proceedings of the 6th Colloquium on Automata, Languages and Programming, 1979.

[13] C. Coit, S. Staniford and J. McAlerney, Towards faster string matching for intrusion detection. In: Proceedings of DARPA Information Survivability Conference and Exhibition, 2002.

[14] R. Sidhu and V. K. Prasanna, Fast regular expression matching using FPGAs. In: Proceedings of IEEE FCCM, 2001.

[15] C. R. Clark and D. E. Schimmel, Efficient reconfigurable logic circuit for matching complex network intrusion detection patterns. In: Proceedings of IEEE FPL, 2003.

[16] J. Moscola, J. Lockwood, R. P. Loui and M. Pachos, Implementation of a content-scanning module for an internet firewall. In: Proceedings of IEEE FCCM, 2003.

[17] C. R. Clark and D. E. Schimmel, Scalable pattern matching on high-speed networks. In: Proceedings of IEEE FCCM, 2004.

[18] I. Sourdis and D. Pnevmatikatos, Pre-decoded CAMs for efficient and high-speed NIDS pattern matching. In: Proceedings of IEEE FCCM, 2004.

[19] Xilinx Vertex-5 family overview. http://www.xilinx.com, 2009.

[20] 黄昆, 张大方, 谢高岗, 金军航, 一种面向深度数据包检测的紧凑型正则表达式匹配算法. 中国科学: 信息科学, 2010, 40(2): 356-370.

[21] B. C. Brodie, R. K. Cytron and D. E. Taylor, A scalable architecture for high-throughput regular-expression pattern matching. In: Proceedings of ACM ISCA, 2006.

[22] J. van Lunteren, High performance pattern-matching for intrusion detection. In: Proceedings of IEEE INFOCOM, 2006.

[23] T. Song, W. Zhang and D. Wang, A memory efficient multiple pattern matching architecture for network security. In: Proceedings of IEEE INFOCOM, 2008.

[24] S. Dharmapurikar and J. Lockwood, Fast and scalable pattern matching for content filtering. In: Proceedings of ACM ANCS, 2005.

[25] H. Lu, K, Zheng, B. Liu, X. Zhang and Y. Liu, A memory-efficient parallel string matching architecture for high-speed intrusion detection. IEEE Journal on Selected Areas in Communication, 2006, 34(10): 1793-1804.

[26] M. Alicherry, M. Muthuprasanna and V. Kumar, High speed pattern matching for network IDS/IPS. In: Proceedings of IEEE ICNP, 2006.

[27] N. Hua, H. Song and T. V. Lakshman, Variable-stride multi-pattern matching for scalable deep packet inspection. In: Proceedings of IEEE INFOCOM, 2009.

[28] L. Tan, B. Brotherton and T. Sherwood, Bit-split string-matching engines for intrusion detection and prevention. ACM Transactions on Architecture and Code Optimization, 2006, 3(1): 3-34.

[29] Y. E. Yang, V. K. Prasanna and C. Jiang, Head-body partitioned string matching for deep packet inspection with scalable and attack-resilient performance. In: Proceedings of IEEE IPDPS, 2010.

[30] S. Dharmapurikar, P. Krishnamurthy, T. S. Sproull and J W. Lockwood, Deep packet inspection using parallel bloom filters. IEEE Micro, 2004, 24(1): 52-61.

[31] N. Tuck, T. Sherwood, B. Calder and G. Verghese, Deterministic memory-efficient string matching algorithms for intrusion detection. In: Proceedings of IEEE INFOCOM, 2004.

[32] A. Broder and M. Mitzenmacher, Network applications of Bloom filters: a survey. Internet Mathematics, 2004, 1(4):485-509.

[33] M. Vutukuru, H. Balakrishna, and V. Paxson, Efficient and robust TCP stream normalization. In: Proceedings of IEE Symposium on Security and Privacy, 2008.

[34] C. Kanich, C. Kreibich, K. Levchenko, B. Enright, G. M. Voelker, V. Paxson and S. Savage, Spamalytics: an empirical analysis of spam marketing conversion.

Communications of ACM, 2009, 52(9): 99-107.

[35] D. Brumley, J. Newsome, D. Song, H. Wang and S. Jha, Towards automatic generation of vulnerability-based signatures. In: Proceedings of IEE Symposium on Security and Privacy, 2006.

[36] R. Sommer and V. Paxson, Enhancing byte-level network intrusion detection signatures with context. In: Proceedings of ACM Computer and Communication Security, 2003.

[37] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley and J. Turner, Algorithms to accelerate multiple regular expressions matching for deep packet inspection. In: Proceedings of ACM SIGCOMM, 2006.

[38] S. Kumar, J. Turner and J. Williams, Advanced algorithms for fast and scalable deep packet inspection. In: Proceedings of ACM ANCS, 2006.

[39] M. Becchi and S. Cadambi, Memory-efficient regular expression search using state merging. In: Proceedings of IEEE INFOCOM, 2007.

[40] R. Smith, C. Estan and S. Jha, XFA: faster signature matching with extened automata. In: Proceedings of IEEE Symposium on Security and Privacy, 2008.

[41] R. Smith, C. Estan, S. Jha and S. Kong, Deflating the big bang: fast and scalable deep packet inspection with extened finite automat. In: Proceedings of ACM SIGCOMM, 2008.

[42] R. Sommer, V. Paxson and N. Weaver, An architecture for exploiting multi-core processors to parallelize network intrusion detection prevention. Concurrency and Computation: Practice and Experience, 2009, 21:1255-1279.

[43] C. R. Meiners, J. Patel, E. Norige, E. Torng and A. Liu,. Fast regular expression matching using small TCAMs for network intrusion detection and prevention systems. In: Proceedings of USENIX Security, 2010.

[44] A. Bavier, N. Feamster, M. Huang, L. Peterson and J. Rexford, In VINI veritas: realistic and controlled network experimentations. In: Proceedings of ACM SIGCOMM, 2006.

Author Biographies:

Kun Huang: Postdoctoral Researcher, Network Technology Research Center, Institute of Computing Technology, Chinese Academy of Sciences, huangkun09@ict.ac.cn

Gaogang Xie: Director and Professor, Network Technology Research Center, Institute of Computing Technology, Chinese Academy of Sciences, Ph.D. Supervisor

Note: Figure translations are in progress. See original paper for figures.