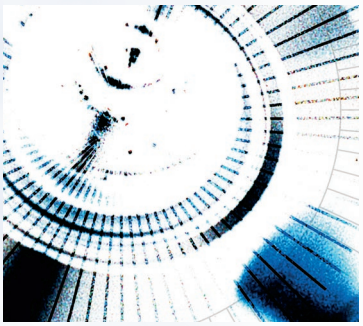
Using String Matching for Deep Packet Inspection

String matching has sparked renewed research interest due to its usefulness for deep packet inspection in applications such as intrusion detection, virus scanning, and Internet content filtering. Matching expressive pattern specifications with a scalable and efficient design, accelerating the entire packet flow, and string matching with high-level semantics are promising topics for further study.



Po-Ching Lin, Ying-Dar Lin, and Tsern-Huei Lee
National Chiao Tung University

Yuan-Cheng Lai
National Taiwan University of Science and Technology

A classical algorithm for decades, string matching has recently proven useful for deep packet inspection (DPI) to detect intrusions, scan for viruses, and filter Internet content. However, the algorithm must still overcome some hurdles, including becoming efficient at multigigabit processing speeds and scaling to handle large volumes of signatures.

Before 2001, researchers in packet processing were most interested in *longest-prefix matching* in the routing table on Internet routers and *multifield packet classification* in the packet header for firewalls and quality-of-service applications.[1] However, DPI for various signatures is now of greater interest.

Intrusion detection, virus scanning, content filtering, instant-messenger management, and peer-to-peer identification all can use string matching for inspection. Much work has been done in both algorithm design and hardware implementation to accelerate the inspection, reduce pattern storage space, and efficiently handle regular expressions.

According to our survey of recent publications about string matching from IEEE Xplore (http://ieeexplore.ieee.org) and the ACM digital library (http://portal.acm.org/dl.cfm), researchers formerly were more interested in pure algorithms for either theoretical interest or general applications, while algorithms for DPI have attracted more attention lately. Likewise, to meet the demand for higher processing speeds, researchers are focusing on hardware implementation in application-specific integrated circuits and field-programmable gate arrays, as well as parallel multiple processors. Since 2004, ACM and IEEE publications have featured 34 articles on ASICs and FPGAs compared to nine in the 1990s and nine again between 2000 and 2003. ACM and IEEE publications have published 10 articles on multiple processors since 2004, with 10 published during the 1990s, and three between 2000 and 2003.

0018-9162/08/$25.00 © 2008 IEEE

Characteristics of String-Matching Algorithms

Researchers can evaluate string-matching algorithms based on the following characteristics:

- *Number of searches*. Some applications, such as search engines, search the same text many times for different querying strings. Building an indexing data structure from the text in advance is therefore worthwhile to perform with the time complexity as low as O(*m*). In contrast, the applications in networking and biological sequences search throughout online text only once without the indexing structure, and the time complexity is linear in *n*.
- *Text compression*. Some algorithms can directly search the compressed text with minimum (or no) decompression, while others scan over the plaintext.
- *Matching criteria*. A match can be exact or approximate. An exact match demands that the pattern and matched text be identical, while an approximate match allows a limited number of differences between them.
- *Time complexity*. Some algorithms have deterministic linear time complexity, while others can have sublinear time complexity by skipping characters not in a match. The latter might be faster on average, but not in the worst case.
- *Number of patterns*. An algorithm can scan one pattern or multiple patterns simultaneously.
- *Expressiveness in pattern specifications*. Pattern specifications range from fixed strings to regular expressions in various syntax options. In addition to primitive notations of alternation, catenation, and Kleene closure, extensions in the syntax of regular expressions include the Unix representations, the extended forms in Posix 1003.2, and Perl Compatible Regular Expression.[1] An increasing number of signatures is specified in regular expressions for their expressiveness.

Reference

1. J. Friedl, *Mastering Regular Expressions*, 3rd ed., O'Reilly, 2006.

DEVELOPMENT OF STRING-MATCHING ALGORITHMS

The "Characteristics of String-Matching Algorithms" sidebar summarizes the characterization and classification of these algorithms. In DPI, *automaton*, *heuristic*, or *filtering* approaches are common. Bit parallelism techniques are often used in computational biology, but rarely in networking. We assume the text length to be *n* characters and the pattern length (or the shortest length in the case of multiple patterns) to be *m* characters.

Automaton-based approach

An automaton-based approach tracks partially matched patterns in the text by state transition in either a *deterministic finite automaton* or a *nondeterministic finite automaton* implementation that accepts the strings in the pattern set. A DFA implementation generally has lower time complexity but demands more space for pattern storage, while an NFA implementation is the opposite.[2] The automaton-based approach is popular in DPI for two reasons:

- The deterministic execution time guarantees the worst-case performance even when algorithmic attacks deliberately generate text to exploit an algorithm's worst-case scenario.
- Building an automaton to accept regular expressions is systematic and well-studied.

Given the wide data bus of 32 or 64 bits in modern computer architectures, tracking the automaton with one input character at a time poorly utilizes the bus width and degrades throughput. Extending the transition table to store transitions for two or more characters is plausible, but it's impractical without proper table compression. Storing a large pattern set is also memory-consuming due to the large number of states. Recent research therefore tries to reduce data-structure space and simultaneously inspect multiple characters. A compact data structure in a software implementation also increases performance due to the good cache locality.

Reducing sparse transition tables. A transition table is generally sparse because most states, particularly those away from the root state, have only a few valid next states. We can compress the table by storing only links to valid next states after one or more input characters and failure links of each state. We also can store the state transition table, the failure links, and the lists of matched patterns in the final states separately in a software implementation to improve the cache locality during tracking.

Snort (www.snort.org), a popular open source intrusion-detection package, has carefully tuned the data structure in this way to improve cache performance. The latest revision uses a basic NFA construction as the default search method (src/sfutil/bnfa_search.c in the source tree of Snort 2.6.1).

Reducing transitions. With the extended ASCII alphabet, an automaton has a maximum of 256 transitions from a state. Splitting an automaton into several smaller ones at the bit level can reduce the number of transitions. For example, suppose the automaton is split into eight, and then one

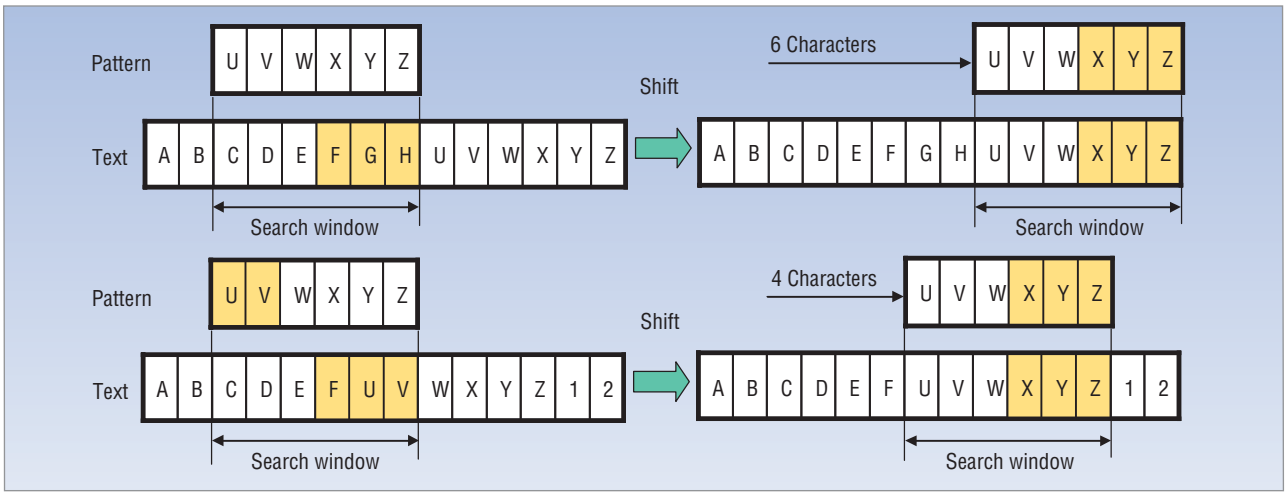


Figure 1. A simple heuristic demonstrates one pattern to visualize why skipping is efficient.

automaton is fed with b7, one is fed with b6, and so on, where b7b6 … b0 denotes the eight bits of the input characters.

This method is implemented in hardware to efficiently track these automata in parallel. These automata are compact because each state has at most two valid transitions for input bits of 0 and 1. Expanding the automata to read multiple characters at a time is also facilitated due to the significantly reduced fanout—in this example, perhaps only 16 valid transitions from a state for four input characters at once.

Because groups of states in an automaton generally have common outgoing transitions that lead to the same set of states for the same input characters, the delayed input DFA (D^2FA) method can effectively reduce these common transitions. A state in a group can maintain only its unique transitions and make a default transition to the state in the group responsible for the common transitions. This method claims to reduce more than 95 percent of transitions for regular expressions on practical products and tools.

Hash tables. A hash table can store the transitions from the states in an automaton to their corresponding valid next states (or failure links) after several input characters. Tracking multiple characters at a time becomes a table lookup. Because only a few input characters can lead to valid next states, the hash table size is still manageable. A filtering approach can weed out unsuccessful searches in the hash table to further accelerate this method. Ternary content addressable memory is an alternative for a table lookup.

Rewriting and grouping. Some combinations of wildcards and repetitions in regular expressions will generate a complex automaton that grows exponentially.[2] It's possible to rewrite the regular expressions to simplify the automaton because we don't have to find every match in the text in some networking applications. Finding an appearance of certain signatures suffices. For example, every string *s* identified by "ab+" (+ denotes one or more) can be identified by "ab" as s itself or a prefix of *s*, so reporting a match against "ab" is sufficient to report an appearance of "ab+".

Furthermore, compiling all the regular expressions in a single automaton can result in a complex automaton. In a multiprocessing environment, we can group regular expressions in separate automata according to the interaction between them. For example, grouping regular expressions sharing the same prefix can merge common states of the prefix and save the storage. An individual processing unit then processes each automaton.

Hardwiring regular expressions. Some designs use building blocks on the FPGA to match patterns from fixed strings to regular expressions. The implementation typically prefers an NFA to a DFA because an NFA has fewer states, and the inherent concurrency of hardware can easily track multiple active states.

A few techniques can reduce the area cost of building blocks. For example, identical substrings from different patterns can share common blocks. Specific hardware logics can directly handle notations in regular expressions such as class of characters, repetitions, wildcard characters, and so on.

Heuristic-based approach

A heuristic-based approach can skip characters not in a match to accelerate the search according to certain heuristics. During the search, a search window of *m* characters covers the text under inspection and slides throughout the text. A heuristic can check a block of characters in the window suffix for its appearance in the patterns. It determines whether a suspicious match occurs and moves to the next window position if not.

Shift values. Because the positions or shift values corresponding to possible blocks are computed and stored in a table beforehand, a table lookup drives shifting the search window in the search stage. Figure 1 illustrates a simple but generic heuristic for only one pattern to visu-

alize why skipping is efficient. In the upper part, because "FGH" is not a substring of the pattern and its suffix is not a prefix of the pattern, shifting the search window by $m = 6$ characters without examining the remaining characters in the window won't miss a match. After the shift, "XYZ" becomes the suffix of both the pattern and the window, meaning a suspicious match occurs. The entire window is then verified, and a match is found.

However, if a suffix of the block is the prefix of some pattern, the shift value should be less than m because the suffix might be the prefix of that pattern after the shift. Figure 1 illustrates this case. We can easily extend this heuristic to handle patterns shorter than the block size. If a short pattern is a substring of the block, looking up the block can claim a match. In addition to the heuristic for matching fixed strings, Gonzalo Navarro and Mathieu Raffinot presented a heuristic to skip text characters for regular-expression matching.[3]

Ideally, most shift values are equal or close to the pattern length m, so the time complexity is sublinear: $O(n/m)$. However, the time complexity could be $O(nm)$ in the worst case, in which the system examines each entire search window after a shift of one character. Although methods exist to guarantee the linear worst-case time complexity for a single fixed string or regular expression,[3,4] they're rarely adopted in DPI, which looks for multiple patterns. Finding an inexpensive solution to achieve sublinear time while ensuring the performance in the worst case would be an interesting challenge to the research community.

Due to their vulnerability to algorithmic attacks, heuristic-based algorithms usually are not preferable for network-security applications.

Due to their vulnerability to algorithmic attacks, heuristic-based algorithms usually are not preferable for network-security applications because an attacker might manipulate the text to degrade performance. Because applications such as Snort have short patterns of only one or two characters, the small value of m makes the advantage of skipping marginal. Nevertheless, for applications with long patterns such as the signatures of nonpolymorphic viruses in the ClamAV antivirus package (www.clamav.net), skipping over the text is still helpful.

Implementation details. Block size, mapping from the blocks to derive shift values, and other implementation details can significantly affect practical performance. When choosing proper parameter values, considerations include the size of the pattern set, block distribution, cache locality, and verification frequency. For example, a large block has fewer chances to appear in the patterns, resulting in less frequent verification.

However, a large block also generally implies a large table that stores shift values mapped from a large number of possible blocks, resulting in reduced cache locality. Careful experimenting should properly tune these parameters. When suspicious matches frequently appear, implementing an efficient method to identify the matched pattern is also important.

Because the block distribution might be nonuniform in practice, some blocks might appear more often than expected, shortening the shift distance and increasing the verification frequency. Checking the matches in additional blocks within the search window can reduce the frequency of verification.

Using a heuristic similar to that in Figure 1 to look for the longest suffix of the search window that's also a substring of some pattern might result in long shift distance even with nonuniform block distribution. However, longer shift distance doesn't always imply better performance. The overhead due to the extra examination must be carefully evaluated.

Filtering-based approach

A filtering-based approach searches text for necessary pattern features and quickly excludes the content not containing those features. For example, if a packet misses any two-character substrings of a pattern, the packet must not have that pattern. Because the efficiency relies on assuming that the signatures rarely appear in normal packets, this approach might suffer from algorithmic attacks if the attacker carefully manipulates the text.

Text filtering. A common method of text filtering is the Bloom filter, characterized by a bit vector and a set of k hash functions $h_1, h_2, \ldots, h_k$ mapped to that vector. When multiple patterns are present, the patterns of a specific length are stored in a separate Bloom filter by setting to 1 the bits the patterns' hash values address. The search queries the set of Bloom filters by mapping the substrings in the text under inspection to them with the same set of hash functions. Specifically, a substring x under inspection is mapped to the Bloom filter storing the patterns of length $|x|$.

If one of the bits in $h_1(x), h_2(x), \ldots, h_k(x)$ isn't set to 1, x certainly isn't in the pattern set; otherwise, x might be in the pattern set, and we must further verify the match. The uncertainty comes from different patterns setting checked bits. The false-positive rate is a function of the bit-vector size, the number of patterns, and the number of hash functions. Properly controlling these parameters can reduce the false-positive rate.

Parallel queries. Parallel queries to the Bloom filters generally are implemented in hardware for efficiency, but efficient software implementation of sequential queries is also possible. For example, the implementation can sequentially query with a set of hash functions, from simple to complex ones, to look for pattern prefixes of a certain length and verify a match if a prefix is found. The simple hash functions are designed to be

rapidly computed and can filter most of the text, so the search is still fast.

If there is a wide range of pattern lengths, there might be many Bloom filters because each length requires one. One solution is to limit the maximum pattern length allowed and break a long pattern into short ones. If all substrings of a long pattern appear contiguously and in order, that pattern is present.

The filtering-based approach doesn't directly support some notations in regular expressions such as wildcards and repetitions. An indirect solution is to extract the necessary substrings from the regular expressions, searching for them and verifying the match if these substrings appear. For example, ClamAV divides the signatures of polymorphic viruses into ordered parts (substrings of the signatures) and tracks the orders and positions of these parts (with a variant of the Aho-Corasick algorithm) in the text to determine whether a signature occurs. Table 1 summarizes the key methods as well as the pros and cons of each.

Table 1. Summary of approaches to string matching for DPI.

Automaton-based
Pros: Deterministic linear execution time, direct support of regular expressions
Cons: Might consume much memory without compressing data structure
1. Rewrite and group regular expressions
2. Reduce number of transitions (D^2FA)
3. Hardwire regular expressions on FPGA
4. Track a DFA that accepts the patterns (Aho-Corasick)
5. Reduce sparse transition table (Bitmap-AC, BNFA in Snort)
6. Reduce fanout from the states (split automata)
7. Track multiple characters at a time in an NFA (JACK-NFA)
Heuristic-based
Pros: Can skip characters not in a match, sublinear execution time on average
Cons: Might suffer from algorithmic attacks in the worst case
1. Get shift distance using heuristics based on the automaton that recognizes the reverse prefixes of a regular expression (RegularBNDM)
2. Get shift distance from fixed block in suffix of search window (Wu-Manber)
3. Get shift distance from the longest suffix of search window (BG)
Filtering-based
Pros: Memory efficient in the bit vectors
Cons: Might suffer from algorithmic attacks in the worst case
1. Extract substrings from regular expressions, filter text with them (MultiFactRE)
2. Filter with a set of Bloom filters for different pattern lengths
3. Filter with a set of hash functions sequentially (Hash-AV)

CURRENT TRENDS IN DPI

Matching expressive pattern specifications with a scalable and efficient design, accelerating the entire packet flow, and string matching with high-level semantics are promising topics for further study.

Matching expressive pattern specifications

Expressive pattern specifications, such as regular expressions, can accurately define the signatures. Efficient solutions to matching regular expressions in DPI are therefore attracting considerable interest. Joao Bispo and his colleagues compared several designs for regular expression matching.[5] Most of these designs can perform regular expression matching on the order of several gigabits per second.

Commercial products, including the Cavium Octeon MIPS64 processor family (www.cavium.com/OCTEON_MIPS64.html), SafeNet Xcel 4850 (http://cn.safenet-inc.com/products/safenetchips/index.asp), and Tarari RegEx5 content processor (www.lsi.com/documentation/networking/tarari_content_processors/Tarari_RegEx_Whitepaper.pdf) all claim to support regular expression matching at gigabit rates. String matching, a problem once believed to be a bottleneck, has become less critical given the latest advances.

Most existing research aims at intrusion-detection applications, especially Snort, which has thousands of signatures, but antivirus applications such as ClamAV claim a signature set of more than 180,000 patterns to date. We believe a more scalable and efficient design for matching a huge set of expressive patterns deserves further study. Moreover, some patterns might belong to only a specific protocol or file type, and some are significant only when they appear in specified positions of the text.

Rather than assuming a simple model of searching for the whole pattern set throughout the entire text, a design can optimize the performance of additional information. An efficient software implementation for these cases is also desirable, since hardware accelerators aren't always affordable in practical applications.

Accelerating packet content processing

Although numerous research efforts have been dedicated to string matching, packet processing in DPI involves even more effort. Vern Paxson and colleagues described the insufficiency of string matching in intrusion detection due to its stateless nature[6] and envisioned a framework of architecture that attempts to exploit the parallelism in network analysis and intrusion detection for acceleration.

Similarly, virus-scanning applications might reassemble packets, unpack and decompress file archives, and handle character encoding before scanning a transferred file. Accelerating only one stage is insufficient due to Amdahl's law. Meeting the high-speed demand in

networking applications requires an integrated architecture with hardware-supported functions.

Commercial products are on this track. For example, the Cavium Octeon MIPS64 processor family includes a TCP unit, a compression/decompression engine, and 16 regular expression engines on a single chip, and claims performance of up to 5 Gbps for regular expression matching plus compression/decompression.

Parsing content in high-level semantics

String matching in network applications might refer to contextual information parsed from high-level semantics.[7] For example, some patterns are significant only within the uniform resource indicators. Spam and Web filtering also demand high-level semantics to analyze the content, as does XML processing.[8] String matching with high-level semantic extraction and analysis from the text is therefore beneficial.

For example, because the Tarari random access XML content processor (www.lsi.com/documentation/networking/tarari_content_processors/Tarari_RAX_Whitepaper.pdf)can help applications directly access information inside XML documents without parsing, it accelerates XML applications significantly. The acceleration of semantic extraction from the text (perhaps with hardware support) and matching patterns with the semantic contextual information is worth studying, and will be helpful for numerous network applications.

Despite existing research, the study of string matching for DPI still has a way to go in the near future. In addition to the growing set of increasingly expressive patterns that makes scalability a challenge, matching with semantically contextual information also complicates the traditional model of string matching that looks for patterns in the text. Dealing with this complication is particularly significant because many existing efforts still use the traditional model to develop their solutions. After all, DPI applications rely on the packet content semantics to make an effective decision. These complexities require expending more effort to develop a scalable, efficient, and effective string-matching solution for DPI applications.

References

1. P. Gupta and N. McKeown, "Algorithms for Packet Classification," *IEEE Network*, vol. 15, no. 2, Mar./Apr. 2001, pp. 529-551.
2. F. Yu et al., "Fast and Memory-Efficient Regular Expression Matching for Deep Packet Inspection," *Proc. Symp. Architectures Networking and Comm. Systems* (ANCS 06), ACM Press, 2006, pp. 93-102.
3. G. Navarro and M. Raffinot, "New Techniques for Regular Expression Searching," *Algorithmica*, Springer-Verlag, vol. 41, no. 2, 2004, pp. 89-116.
4. Z. Galil, "On Improving the Worst-Case Running Time of the Boyer-Moore String Searching Algorithm," *Comm. ACM*, vol. 22, no. 9, 1979, pp. 505-508.
5. J. Bispo et al., "Regular Expression Matching for Reconfigurable Packet Inspection," *Proc. IEEE Int'l Conf. Field-Programmable Technology* (FPT 06), IEEE Press, 2006, pp. 119-126.
6. V. Paxson et al., "Rethinking Hardware Support for Network Analysis and Intrusion Prevention," *Proc. Usenix Workshop Hot Topics in Security*, Usenix, 2006, pp. 63-68; http://imawhiner.com/csl/usenix/06hotsec/tech/paxson.html.
7. R. Sommer and V. Paxson, "Enhancing Byte-Level Network Intrusion Detection Signatures with Context," *Proc. ACM Computer and Comm. Security* (CCS 03), ACM Press, 2003, pp. 262-271.
8. T.J. Green et al., "Processing XML Streams with Deterministic Automata and Stream Indexes," *ACM Trans. Database Systems*, Dec. 2004, pp. 752-788.

Po-Ching Lin *is a PhD candidate in the Department of Computer Science at the* ***National Chiao Tung University*** *in Hsinchu, Taiwan. His research interests include network security, string-matching algorithms, hardware-software codesign, content networking, and performance evaluation. Lin received an MS in computer science from the National Chiao Tung University. He is a student member of the IEEE. Contact him at pclin@cis.nctu.edu.tw.*

Ying-Dar Lin *is a professor in the Department of Computer Science at* ***National Chiao Tung University****. His research interests include design, analysis, implementation, and benchmarking of network protocols and algorithms; wire-speed switching and routing; and embedded hardware-software codesign. Lin received a PhD in computer science from the University of California, Los Angeles. He is a senior member of the IEEE and a member of the ACM. Contact him at ydlin@cs.nctu.edu.tw.*

Yuan-Cheng Lai *is an associate professor in the Department of Information Management at* ***National Taiwan University of Science and Technology*** *in Taipei, Taiwan. His research interests include high-speed networking, wireless network and network performance evaluation, Internet applications, and content networking. Lai received a PhD in computer science from the National Chiao Tung University. Contact him at laiyc@cs.ntust.edu.tw.*

Tsern-Huei Lee *is a professor in the Department of Communications Engineering at the* ***National Chiao Tung University****. His research interests include high-speed networking, broadband switch systems, network flow control, data communications, and string-matching algorithms. Lee received a PhD in electrical engineering from the University of Southern California. Contact him at thlee@banyan.cm.nctu.edu.tw.*