2ND QUARTER 2006, VOLUME 8, NO. 2

IEEE COMMUNICATIONS SURVEYS
The Electronic Magazine of Original Peer-Reviewed Survey Articles

www.comsoc.org/pubs/surveys

PROFILING AND ACCELERATING STRING MATCHING ALGORITHMS IN THREE NETWORK CONTENT SECURITY APPLICATIONS

PO-CHING LIN, ZHI-XIANG LI, AND YING-DAR LIN, NATIONAL CHIAO TUNG UNIVERSITY
YUAN-CHENG LAI, NATIONAL TAIWAN UNIVERSITY OF SCIENCE AND TECHNOLOGY
FRANK C. LIN, CISCO SYSTEMS, INC.

ABSTRACT

The efficiency of string matching algorithms is essential for network content security applications, such as intrusion detection systems, anti-virus systems, and Web content filters. This work reviews typical algorithms and profiles their performance under various situations to study the influence of the number, the length, and the character distribution of the signatures on performance. This profiling can reveal the most efficient algorithm in each situation. A fast verification method for some string matching algorithms is also proposed. This work then analyzes the signature characteristics of three content security applications and replaces their original algorithms with the most efficient ones in the profiling. The improvement for both real and synthetic sample data is observed. For example, an open source anti-virus package, ClamAV, is five times faster after the revision. This work features comprehensive profiling results of typical string matching algorithms and observations of their application on network content security. The results can enlighten the choice of a proper algorithm in practical design.

Detecting and filtering intrusions, worms, viruses, and inappropriate Web pages involves *string matching* for designated signatures in the application content, as opposed to *packet classification* that matches fixed fields in the packet header [1]. The position and length of the signatures in the packet payload are unknown beforehand, so scanning throughout the packet payload for signatures is normally less efficient than packet classification. String matching is reported to be a bottleneck for network content security applications [2–5], and thus its efficiency is critical.

Signatures in content security applications are typically represented as patterns in some forms. For clarifying the terminology, a string is a sequence of characters, and a pattern is an occurrence of a string in the text [6]. Signature characteristics in different applications may vary wildly in the number, length, and character distribution in the alphabet. For instance, anti-virus systems feature a large number of long signatures, while the intrusion detection systems may have short signatures of one or two characters. No existing string matching algorithms can search for signatures of various characteristics faster than others can. The choice of a proper algorithm therefore becomes important in the application design.

This work reviews existing string matching algorithms and their applications in network content security. The characteristics of signatures in three typical open source packages are investigated: ClamAV (http://www.clamav.net) for anti-virus, DansGuardian (http://dansguadian.org) for Web content filtering, and Snort (http://www.snort.org) for intrusion detection systems (IDS). This work profiles the performance of various algorithms. The most efficient algorithm for each package is identified and then implemented on each package. These revised packages are benchmarked and the improvement in performance for sample sets of both synthetic and real data is demonstrated. The impact of memory and cache accesses on performance is also measured quantitatively.

In addition, this work proposes *classified* RKBT (Rabin-Karp with binary search and two-level hashing) to accelerate the performance of the original RKBT algorithm [7] for a large pattern set. The RKBT algorithm can verify a possible match found by some string matching algorithms [8], and its

efficiency becomes critical as the number of possible matches increases.

This work focuses on string matching algorithms that apply to the field of network content security. For general discussions of string matching algorithms and their applications in other fields, the readers are referred to some recent textbooks, such as [6] and [9]. Moreover, a number of algorithms can be accelerated by hardware parallelism. A comprehensive survey of these hardware acceleration mechanisms deserve the writing of another paper and is beyond the scope of this work. Interested readers are referred to a brief survey in [10], and some commercial solutions, such as SafeNet SafeXcel 4850 [11] or Tarari RegEx Content Processor [12].

The main contributions of this work are summarized as follows.

- This work offers a comprehensive survey as well as the profiling results of typical string matching algorithms at various aspects and their application on network content security. The results can enlighten the choice of a proper algorithm in practical design.
- This work suggests the most efficient algorithm for each network content security package and demonstrates the performance gain for both synthetic and real data through implementation and benchmarking.
- A new verification method, classified RKBT, is proposed to accelerate some string matching algorithms.

The rest of this work is organized as follows. We review typical algorithms and three open-source network content security packages. We will discuss verification in some algorithms and propose a new method for verification. We present the profiling results and identify the most efficient algorithm for each application. The performance improvement for both synthetic and real data after revising the packages is also demonstrated. We then conclude the study.

Related Works

Typical String Matching Algorithms

The single string matching problem is to search for all the occurrences of a string p, called the pattern, in the text $T = t_1t_2t_3\ldots t_n$ on the same alphabet $\subseteq$, where n is the length of the text. Multiple string matching extends the problem to search for the pattern set $P = \{p^1, p^2 \ldots, p^r\}$ simultaneously in the text. During the search, a search window of the pattern length is moved along the text, and the pattern is searched for within the window. Pattern matching can be exact or approximate. An *exact matching* algorithm stipulates that the pattern and the matched text should be *exactly* the same, while an *approximate matching* algorithm allows a limited error between the pattern and the matched text. This work concentrates on only the former because the majority of the content security applications use it to find out the signatures. A tutorial on the approximate matching algorithms can be found in Navarro [13].

Exact string matching algorithms can be categorized in various ways. One way of categorization is grouping them into three general approaches: prefix searching, suffix searching, and factor searching, depending on which part of the pattern is searched for within the search window [6, 14]. A string X is the prefix, suffix, and factor of XY, YX, and YXZ, respectively, where Y and Z are also strings. The time complexity of the algorithms can be linear or sub-linear. A sub-linear time algorithm is feasible by skipping characters that do not need to be examined in the text.

This work categorizes the algorithms into four categories to emphasize the data structure that drives the matching. These categories are automaton-based, heuristics-based, hashing-based, and bit-parallelism-based. An automaton-based algorithm builds a finite state automaton from the patterns in the preprocessing stage and tracks the partial match of the pattern prefixes in the text by state transition in the automaton. A heuristics-based algorithm allows skipping some characters to accelerate the search according to certain heuristics. Some algorithms require a verification algorithm following a possible match to verify if a true match occurs. A hashing-based algorithm compares the hash values of characters in the text segment by segment with those of the characters in the patterns. If both hash values are equal, a possible match may occur. The characters in the text and those in the patterns are then compared to verify if a true match occurs. A bit-parallelism-based algorithm simulates the operation of a non-deterministic finite automaton that tracks the partial match of the prefix or the factor of the pattern by means of the parallel bit operations inside a computer register word in which the state transition status is encoded [15]. Table 1 summarizes typical algorithms in these four categories. Each algorithm will be detailed in the following text.

Automaton-based — The Aho-Corasick (AC) algorithm [16] was proposed for multi-pattern matching. A finite automaton that accepts all the strings in the pattern set is built in the preprocessing stage. Each character in the text is then fed sequentially to the automaton that tracks partially matched patterns through state transition, so the time complexity is $O(n)$. If one of the final states is reached, a match is claimed. Although the AC algorithm is theoretically independent of the pattern set size in efficiency, it will become slow for a large pattern set in practice because of the worse cache locality in accessing a large transition table. Effectively compressing the transition table to reduce the memory requirement and enhance the cache locality becomes active research in the implementation of the AC algorithm.

Norton [17] presents the Optimized-AC algorithm to reduces the memory requirement in the well-known IDS package of Snort by compressing the transition table into a compressed sparse vector format or a banded-row format. These variants are generally faster than the standard AC algorithm due to their better cache locality. Some hardware acceleration mechanisms are also proposed for the AC algorithm. We name just a few in this article. Tuck *et al.* [18] use both bitmap and path compression to compress the transition table. This optimization in hardware implementation gains from 31 percent to twice the throughput for Snort rule sets. However, the software implementation is not as good because some inefficient bit operations in this optimization are inefficient in software. Tan and Sherwood [19] split the finite automaton in the AC algorithm into several ones in the bit level, say a set of four automata, each responsible for two bits of an input character. The state transition in each automaton is driven by the individual bits. By parallel tracking of these automata, they claimed a system ten times faster than the best known approaches for the Snort rule set.

Heuristics-based — The Boyer-Moore (BM) algorithm is a single string matching algorithm. It allows skipping over characters that cannot be a match in the text according to two heuristics: the bad-character heuristic and the good-suffix heuristic [20]. Two shift functions are pre-computed according to the two heuristics before the search. The characters within the search window are searched backward from the last character for the longest suffix that is also a suffix of the pattern. The two shift functions are referred to only when a character mismatch is found. The maximum of these two function val-

Algorithms	Approach	Time Complexity	Search Type	Multiple String	Key Ideas
Aho-Corasick	Automaton-based	Linear	Prefix	Yes	Finite automaton that tracks the partial prefix match.
Optimized-AC		Linear	Prefix	Yes	Compress the transition table of the finite automaton in the Aho-Corasick algorithm.
Boyer-Moore	Heuristics-based	Sub-linear	Suffix	No	Bad-character and good-suffix heuristics to determine the shift distance.
Horspool		Sub-linear	Suffix	No	Bad-character heuristics only.
Set-wise Horspool		Sub-linear	Suffix	Yes	Bad-character heuristics for multiple patterns.
Wu-Manber		Sub-linear	Suffix	Yes	Determine the shift distance from a block of characters in the suffix of the search window.
Modified-WM		Sub-linear	Suffix	Yes	Tuning the table size and hash function in the Wu-Manber algorithm.
FNP/FNP^2		Sub-linear	Prefix	Yes	Determine the shift distance from a block of characters in the prefix of the search window and the suffix of the block.
Rabin-Karp	Hashing-based	Linear	Prefix	No	Compare the text and the patterns from their hash functions.
RKBT		Linear	Prefix	Yes	Two-level hashing with binary search.
SOG	Bit-parallelism-based	Linear	Prefix	Yes	Bit-parallelism and q-gram for prefix matching.
BG		Sub-linear	Factor	Yes	Bit-parallelism and q-gram for factor matching.

Table 1. *Categorization of typical string matching algorithms.*

ues is the shift distance of the search window. If no mismatch is found, a pattern match is claimed.

The BM algorithm has the best performance when the shift distance is close to the pattern length. The time complexity is sub-linear of $O(n/m)$ in the best case and $O(n\log_{|\Sigma|}m/m)$ on average [21], where m is the pattern length. However, the worst-case performance can be $O(nm)$ when both the pattern and the text contain the same single character. Some modifications of the BM algorithm, such as Galil's approach [22], can guarantee the worst-case performance of $O(n)$.

Horspool simplified the BM algorithm [23] by resorting to only the bad-character heuristic, which is useful for a reasonably large alphabet, say the ASCII table, because the probability of a character mismatch with the characters of the pattern within the search window is high. Although the BM algorithm can have longer shift distance by taking the maximum of the bad-character function and the good-suffix function, the Horspool algorithm is generally faster in practice because it spends time only in looking up one function, i.e. the bad-character function, in each shift. The simplification compensates for the loss of efficiency due to the shorter shift distance.

The set-wise Horspool algorithm [6] extends the Horspool algorithm to search for multiple patterns simultaneously. The patterns to be searched are represented as a reverse trie. Like the Horspool algorithm, the set-wise version compares the characters backward from the rightmost one of the search window with those in the reverse trie. The search window is shifted along the text in a way similar to the bad-character shift in the Horspool algorithm. A match is claimed if the final state of the trie is reached.

The Horspool algorithm is not scalable for a large pattern set because the rightmost character will be very likely to appear in the patterns and so the average shift distance will be decreased. The Wu-Manber (WM) algorithm [24] improves the set-wise Horspool algorithm by reading a block of B characters rather than only one character for the shift value. The probability that a block of characters appear in the patterns is less than the probability that a single character does, so longer shift distance can be expected. The shift value for each different block (totally $|\Sigma|^B$ different blocks) is stored in a shift table. The memory space of the shift table limits the practical value of B.

The WM algorithm requires that the patterns have the same length. If all the pattern lengths are not equal, only the first l_{min} characters of each pattern are considered, where l_{min} denotes the length of the shortest pattern (LSP). The shift table is built in the preprocessing stage according to the heuristic similar to that in the Horspool algorithm.

1 The shift distance is $l_{min} - B + 1$ when the block of the last B characters in the search window does not appear in any patterns, i.e. not a substring of any patterns. The reason is that any shift shorter than $l_{min} - B + 1$ will locate the block of B characters inside the search window again, contradicting that the block does not appear in any patterns.
2 The shift distance is $l_{min} - j$ otherwise, assuming that the rightmost occurrence of the last B characters in the search window ends in position j of some pattern.

The WM algorithm hashes the block of the last B characters in the search window to look up the shift table for the shift distance during the scanning. The hash function may not be one-to-one so that a shift table with fewer than $|\Sigma|^B$ entries is possible. In this case, the shift value in an entry is

the minimum shift value mapped to that entry. Saving table space and having longer shift distance is a trade-off in the choice of the hash function. A possible match occurs when the shift value is 0. A verification algorithm, left open and not specified by the WM algorithm, follows to verify if a true match occurs. We will discuss the verification algorithms more.

The time complexity of the WM algorithm is sub-linear on average, but depends considerably on l_{min} because the maximum shift distance is $l_{min} - B + 1$. If l_{min} is small, the performance will be degraded significantly.

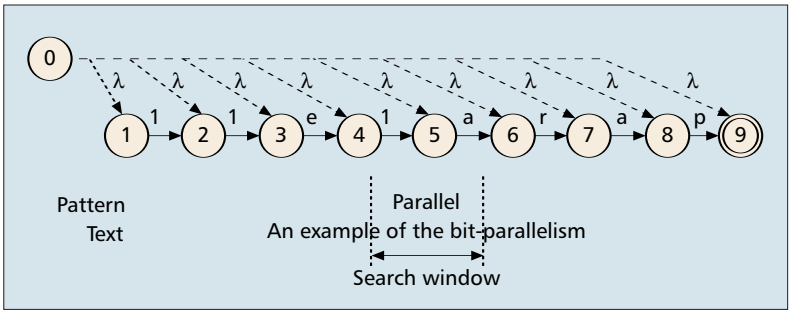

Figure 1. *An NFA recognizing all the suffixes of the reverse pattern of "parallel" in BNDM.*

The Modified-WM algorithm in Snort (see the comments in *mwm.c* under the Snort source tree of */src/sfutil*) is a variant of the original implementation of the WM algorithm. Note that some patterns in Snort are only one character long [25], making the shift distance for a block size of $B \geq 2$ undefined. This variant involves a one- or two-character block and groups all of the patterns with the same hash value from the pattern prefix for verification. The Modified-WM algorithm is more efficient than the original WM implementation, as will be presented later.

The shift distance of $l_{min} - B + 1$ in the WM algorithm can be very short when l_{min} is close to the block size. The FNP/FNP[2] algorithms [25, 26] allow longer shift distance than the WM algorithm by considering the appearance of not only the whole block but also the suffix of the block in the patterns. The observation is simple. For example, if a block "abc" does not appear in any patterns, and both suffixes of "bc" and "c" are not prefixes of any patterns, then the shift distance of three characters is safe, i.e., not missing any possible match. In contrast, the WM algorithm is more conservative because it does not consider the suffix and allows the safe distance of only one character (assuming $l_{min} = 3$ and $B = 3$ in this example). The improvement is particularly significant for a pattern set with short patterns, such as those in Snort. Like the WM algorithm, verification for a true match follows if a partial match within the search window is found.

Hashing-based — The Rabin-Karp (RK) algorithm [27] is designed to handle single string matching. During the search, the hash value of each segment of m characters in the text, $h(t_{s\ldots s+m-1})$, is compared with the hash value of the pattern, $h(p)$, for $s = 1\ldots n-m+1$, where m is the pattern length and h is the hash function. If $h(t_{s\ldots s+m-1}) = h(p)$, a possible match may occur at the position of s. The pattern is then compared with this segment character by character to verify whether a true match occurs or not.

The RK algorithm can be extended for multiple string matching by storing the hash values of the patterns in the pattern set in an ordered table. The comparison of the hash values becomes binary search in the ordered table, to find whether the hash value of a text segment is equal to that of some pattern in the ordered table. However, the binary search requires table accesses on the order of $\log_2 r$ times, where r is the number of patterns. The binary search is costly even for a pattern set of moderate size, say roughly 10 accesses on average for 1,000 patterns.

Muth and Manber [7] use two-level hashing to accelerate the binary search. Besides the ordered table of the first hash values for binary search, a bitmap is constructed from a second hash function for quick indexing. The bitmap indicates whether a match is possible in the ordered table. The binary search is performed only when the bitmap indicates the possibility of a match, and hence the number of binary searches can be reduced. This algorithm is denoted as RKBT herein to stand for the BK algorithm with binary search and two-level hashing. The detailed operation is left to a later section.

Bit-parallelism-based — The Shift-Or (SOR) algorithm [28] and the Backward Nondeterministic DAWG Matching (BNDM) algorithm [29] were proposed for single string matching with bit-parallelism to simulate the tracking of a non-deterministic finite automaton (NFA). The simulation in both algorithms is similar; the difference is the parts in the pattern that are tracked. The former keeps a set of all the *prefixes* of the pattern that have been matched, while the latter keeps the set of all the *factors* that have been matched. We use the BNDM algorithm as the example to explain the bit-parallelism operation. Consider the following NFA that recognizes the suffixes of the reverse pattern of "parallel" in Fig. 1 (λ denotes the null string). Tracking the NFA can search the factors in the pattern of "parallel".

The state of the search is represented by an m-bit computer word $D = d_m \ldots d_1$, initialized by the 1^m to denote the λ-transitions. A table B stores a bit mask $b_1 \ldots b_m$ for each character c in Σ. The bit mask sets the bits b_j to 1 in $B[c]$ if the j-th character of the pattern is c. For example, B['l'] = 00001101 for the pattern 'l'. During the search, the search window is scanned backward from the last character, and D is updated for each character t_s read in the window with the formula $D' \leftarrow D \,\&\, B[t_s]$, followed by $D'' \leftarrow D' << 1$, where "&" denotes the "and" operation. For example, assume D is "11111111" in Fig. 1 in the beginning. D' becomes "01010000" after "a" is read, indicating that two factors of "a" appear in the second and the fourth position of "parallel". D' then becomes "10100000", the next D in the beginning of the next iteration. When "p" is read next, D' becomes "10000000", indicating a factor "pa" in the first position of "parallel". The process goes on until the end of the text. A match is claimed if the entire search window has been searched and $D' = 10^{m-1}$.

Factor-based search also allows skipping characters that cannot be a match. The observation is simple. Assume a suffix μ of the search window is a factor of the pattern, but the suffix βμ is not, where β is a character. The search window can be safely shifted after β. A shift smaller than β cannot get a match because it violates the assumption that βμ is not a factor. Like the WM algorithm, the time complexity of BNDM is also sub-linear time on average.

The SOG and BG algorithms [8] extend the SOR and BNDM algorithms, respectively, for multiple string matching, where G is short for q-Gram, as explained immediately. In these two algorithms, multiple patterns are viewed as a single pattern of classes of characters. The characters in the same position of each pattern are grouped into a class. For example, two patterns "cat" and "dog" are viewed as a single pattern of "[cd][ao][tg]". Therefore, multiple string matching is transformed into single string matching. The SOR or BNDM algorithm searches for the pattern of classes of characters.

Application	Packages	Version	Algorithms[1]	Number of Patterns	Classified[2]	Pattern Length	Char. Set Distribution[3]
Anti-virus	ClamAV	0.85	**Aho-Corasick Wu-Manber**	26467	No	10~210	Type 1
Content-Filter	DansGuardian	2.8.0.4	**Horspool DFA**	5867	No	2~64	Type 2
IDS/IPS	Snort	2.3.3	AC-std AC-Full AC-Sparse AC-Banded AC-SB **Modified-WM** LowMemTrie	Patterns for all groups: 14295 Total rules: 2246	Yes 173 groups Max group size: 1174 Min group size: 12	1~107	Type 1

[1] The default algorithms are marked in bold.

[2] Indicate whether the patterns are classified into several subsets, rather than collapsed in a single set.

[3] Type 1: close to uniform distribution. Type 2: biased to English character set.

■ Table 2. *Selected open source network content security packages.*

False positives, however, may occur in this approach. For example, "dag" can match "[cd][ao][tg]", but it is neither "cat" nor "dog". A possible match should be verified to avoid a false positive. To reduce the number of false positives, each pattern in the pattern set is viewed as an $m - q + 1$ q-gram sequence, where m is the pattern length. The patterns become "ca-at" and "do-og" in the above example. A q-gram expands the effective alphabet size from $|\Sigma|$ to $|\Sigma|^q$ so that the number of false positives in the text and thus verifications can be reduced. Both algorithms use the RKBT algorithm to accelerate the verification.

Other Approaches — String matching algorithms have been developed for decades. Full coverage of all algorithms in this article is nearly impossible. This sub-section covers other algorithms that do not belong to the above categories. For more algorithms and their tutorial, readers are referred to some good online resources, such as [30].

The ExB and E^{2}xB algorithms [31, 32] are designed for intrusion detection. They search for one pattern after another sequentially, and exclude the pattern that is impossible to appear in the packet content. The exclusion is based on this simple observation: If a pattern has a character c, the text must have the character c to contain a match. These two algorithms assume that the intrusion signatures scarcely appear in the real environment, so patterns that cannot be a match can be quickly excluded. The BM algorithm follows if a match is possible, i.e., not excluded. The two algorithms are not scalable to a large pattern set because of their sequential search of patterns. They also may not apply when the patterns appear frequently, a case that violates their assumption. The Set Backward Oracle Matching (SBOM) algorithm [33] uses a factor oracle to recognize at least all the factors in the patterns. It is shown to be most efficient for a small character set, and can better apply to bioinformatics. Some algorithms combine two or more aforementioned algorithms, such as the AC_BM algorithm [34]. This algorithm applies the bad-character and good-suffix heuristics on the automaton in the AC algorithm so that skipping characters in the search window is made possible. However, it has the same scalability problem as the set-wise Horspool algorithm for a large pattern set since the heuristics are based on characters rather than on blocks.

Selected Packages

Table 2 lists the number of patterns, the maximal and minimum pattern lengths, and all supported algorithms in three open-source packages of network content security applications. The details of each package will be explained in the following paragraphs.

ClamAV — ClamAV contains two types of virus patterns: basic patterns that are a simple sequence of characters, and multi-part patterns composed of multiple sub-patterns. The former occupies 93 percent of the total patterns. ClamAV scans basic patterns by the WM algorithm. If no virus is found, the multi-part patterns are then scanned by a variant of the AC algorithm, in which the automaton is represented as a two-level trie [35]. Sub-patterns with a common prefix of two characters are stored in a linked list under the leaf that represents the common prefix. ClamAV uses a table to keep the number of sub-patterns that have been found for each pattern. All sub-patterns of a multi-part pattern must be matched in sequence to assert a virus. ClamAV also supports a simplified form of regular expressions. For example, ClamAV allows "bounded gaps" that specify the minimum and maximum distances allowable between two consecutive sub-patterns by recording the position of a sub-pattern and calculating the distance from its last sub-pattern.

DansGuardian — DansGuardian searches for all keywords in the Web content, and determines whether the content belongs to a banned category. DansGuardian implements the Horspool algorithm and a deterministic finite automata (DFA) algorithm. In the preprocessing, it builds a two-dimensional array, called the graph data, to represent a transition table of the whole DFA that accepts the keywords. Only one copy of redundant keywords from different categories are kept in the pattern set. The graph data is then searched for the nodes that have common prefixes but have fewer than 12 branches (i.e., fewer than 12 keywords from that node). These keywords represented by traversing from the root through these nodes to leaves are moved into another group for searching with the Horspool algorithm one by one. After the Horspool search, DansGuardian continues to search for all the keywords in the graph data. At least 12 keywords share each prefix in the graph data so that traversing the DFA can search for these keywords simultaneously. Finally, Dans-

Guardian determines whether the content should be banned according to the matched keywords. If the *forcequicksearch* option is enabled in the configuration, all the keywords will be searched for with the Horspool algorithm one by one.

Snort — Snort divides its rules into rule subsets associated with unique characteristics in the packet header, such as port numbers, ICMP types, and transport protocol identifiers. The packet header is examined for the unique characteristics first to determine which rule subset to be referred to, and signatures in that rule subset are then searched for [36]. The Modified-WM algorithm searches for the signatures by default. If the signatures in a rule are found, the rest of the rule, represented as options in the rule specification, is verified to claim a true match. If a complete rule match has been found, Snort inserts the rule into the event queue. Finally, Snort processes the event queue and selects a single event for alert. Snort also supports signatures in regular expressions conforming to the specification of Perl Compatible Regular Expressions (PCRE) [37]. PCRE matching is one of the options in the rule specification, and like the other options, it is performed after the search of the rule subset. To accelerate the search, necessary factors of a regular expression are manually added into the rule subset as a hint of possible appearance of the regular expression. For example, "abcd" is a necessary factor of the regular expression "(abc) + d". PCRE matching is performed only when the hint is found, and hence unnecessary PCRE matching can be avoided.

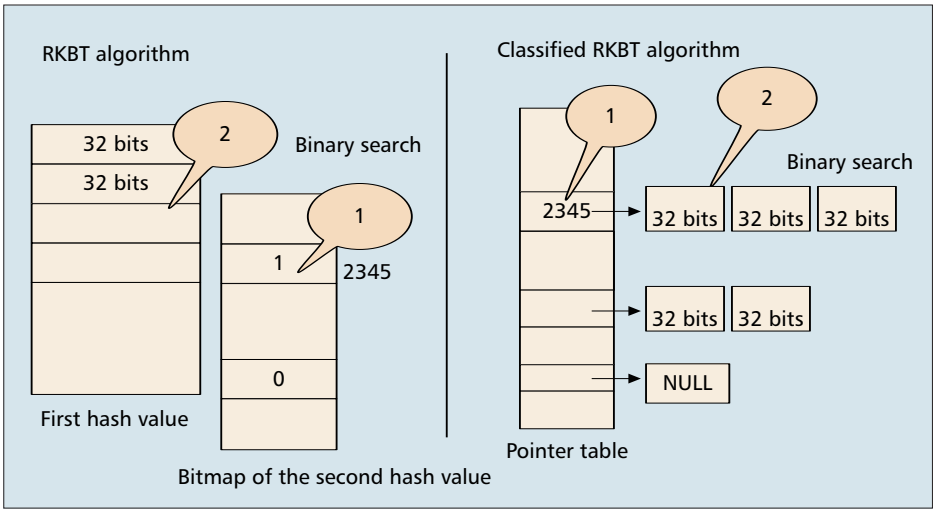

Figure 2. *The operations of the RKBT algorithm (left) and the Classified RKBT algorithm (right).*

The Verification Algorithm

A number of string matching algorithms, such as the BG and WM algorithms, rely on a verification algorithm to assert a true match when a possible match is found, as discussed earlier. The RKBT algorithm can serve the purpose [8], but it can be inefficient for a large pattern set because of its data structure discussed below. Classified RKBT (CRKBT) is proposed herein to accelerate the verification.

The Proposed CRKBT Algorithm

The operation of the RKBT algorithm is illustrated in Fig. 2a. Each pattern in the pattern set is viewed as consecutive blocks of four bytes, and so each block can form a 32-bit integer. If the pattern length is not a multiple of four, the last block is padded with zeros. The first hash function is defined by xor'ing the integers in these blocks. In the preprocessing, an ordered table is constructed to store the first 32-bit hash values of the patterns. A second hash function is derived from the first by xor'ing the lower 16 bits and the upper 16 bits of the first hash values. A bitmap of 2^{16} entries is then built to store the second hash values. The i'th bit of the bitmap is 1 if at least one pattern has i as its second hash value, and is 0 otherwise.

The bitmap indicates whether binary search in the ordered table is necessary. Assume the second hash value of a search window is i in the search stage. If the i'th bit of the bitmap is 0, no possible match will occur and the verification fails. When the i'th bit of the second hash value is 1, say the 2345th bit in Fig. 2, the ordered table is searched for the first hash value with binary search. If the first hash value is found, the characters of the patterns with that value are compared with those in the search window one by one to check if a true match occurs. Otherwise, the verification fails.

The performance of the RKBT algorithm is degraded significantly for a large pattern set. The probability that a bit in the bitmap is set to 1 will be high in such a case. For example, nearly 80 percent of the bits in the bitmap are 1 for 100,000 patterns from the probabilistic estimation. Consequently, binary search in the ordered table of the large pattern set becomes frequent and dominates the verification time.

A classified variant, namely CRKBT, is proposed herein to accelerate the RKBT algorithm. Figure 2b illustrates its operation. CRKBT divides the ordered table for binary search into several small tables associated with the second hash values. The search scope is reduced to only a subset of the patterns that have the same second hash value, so the binary search is much faster. A pointer table replaces the bitmap in RKBT. The i'th pointer points to an ordered table of only patterns that have i as their second hash value, and points to NULL if no patterns have the second hash value of i. The pointer table is looked up when verification is required. If the corresponding pointer of the second hash value is not NULL, the ordered table that the pointer points to is searched with binary search. The overhead of the CRKBT algorithm is 256 KB of the pointer table (2^{16} entries * 4-byte pointer) and at most 256 KB (4-byte integer for the length) to store the lengths of the divided ordered tables. The cost is minor given the memory space in the order of several hundreds MB to GB on modern computers.

Experiments

The execution time of both the RKBT and CRKBT algorithms is benchmarked as follows. The text of 32 MB is randomly generated from the alphabet of 8-bit characters. The patterns are generated from the same alphabet and the shortest pattern length is 8. Both the text and the patterns reside in the main memory in the execution. The tests run on a computer with a 2.8 GHz Pentium 4 processor, 1 GB of memory, and 512 KB cache. Both algorithms are written in C, compiled with the gcc compiler, and run on Linux kernel 2.6.5. Figure 3 presents the benchmark results.

The execution time of both algorithms for small pattern sets is close because of few possible matches and thus few

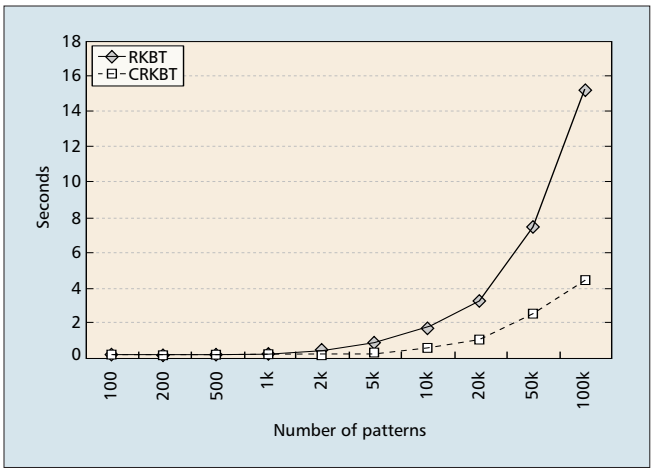

Figure 3. *The execution time of the RKBT and CRKBT algorithms.*

chances of binary search in the ordered table. As the number of patterns increases, the number of possible matches also increases. The difference in execution time becomes obvious. The search scope for binary search is small in the CRKBT algorithm (only the subset of patterns that have the same second hash value), so the binary search is fast. The CRKBT algorithm is four times faster than the RKBT algorithm when the number of patterns grows to 100,000 patterns. Therefore, CRKBT is more scalable to a large pattern set.

We replace the RKBT algorithm with the CRKBT algorithm in the SOG and BG algorithms, denoting the revised versions as the SOG + and BG +. Figure 4 shows both the SOG and BG algorithms are accelerated by this new verification algorithm, particularly for a large pattern set. The BG + algorithm is twice faster than the BG algorithm for the pattern set of 100,000 patterns.

Analysis

Binary search in a large pattern set can dominate the verification time. The CRKBT algorithm reduces the size of the ordered table by dividing the patterns into subsets to accelerate the search. Assume the number of patterns is r. The bitmap or the pointer table is checked first in both algorithms. In RKBT, the probability that a bit is set to one is p, where p can be estimated to be

$$1-\left(\frac{65535}{65536}\right)^{r}.$$

If the bit is set to one, then binary search in the ordered table follows. The expected number of memory accesses is $\log_2 r + 1$ in the table. If the bit is zero, the verification fails. Therefore, the expected number of memory accesses in the RKBT algorithm is $p(\log_2 r + 1) + (1 - p)$. Because the expected size of each ordered table to which a pointer points in the CRKBT algorithm is

$$1+\frac{r}{65536},$$

the expected number of memory accesses in the binary search becomes only

$$\log_2\left(1+\frac{r}{65536}\right)+1,$$

which is much smaller than $\log_2 r + 1$ in the RKBT algorithm for a large r. The expected number of memory accesses in the

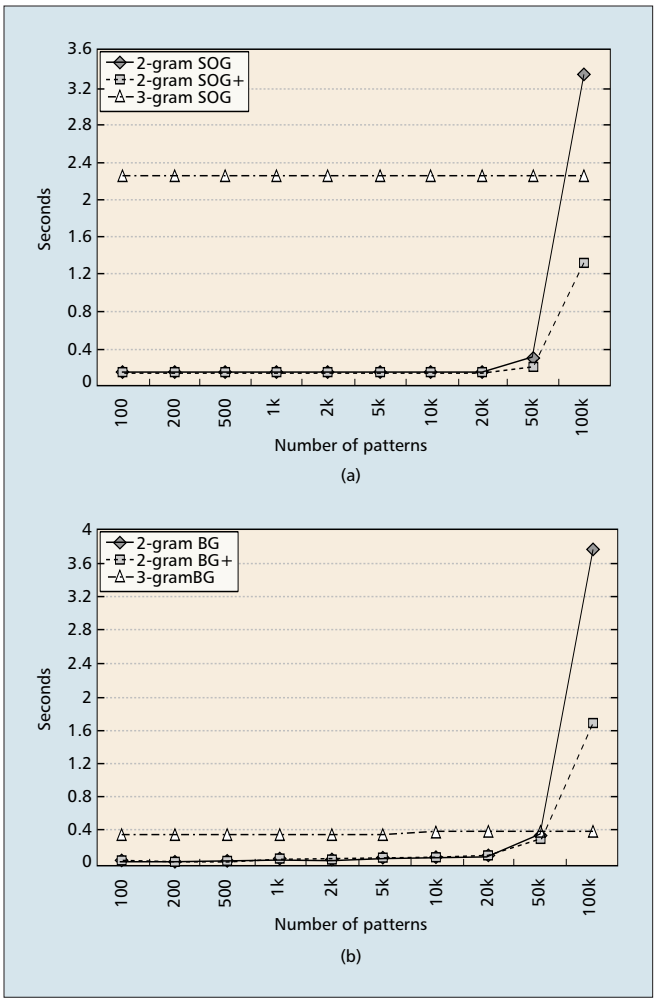

Figure 4. *a) The execution time of the 2-gram SOG and 2-gram SOG+ algorithms; and b) the execution time of the 2-gram BG and 2-gram BG+ algorithm.*

binary search of the CRKBT algorithm is then

$$p\left(\log_2\left(1+\frac{r}{65536}\right)+1\right)+(1-p).$$

Therefore, the CRKBT algorithm is more scalable than the RKBT algorithm.

Profiling Algorithms

In the external and internal profiling, the benchmarking environment and configuration are the same as those described earlier. We select the string matching algorithms from earlier to benchmark their performance for various pattern lengths and pattern set sizes. The CRKBT algorithm is also involved in this benchmark. The implementation of the WM algorithm leverages the code in the Agrep package [38], while that of the AC, Optimized AC, and Modified-WM leverages the code in Snort. The other algorithms are implemented from scratch.

External Profiling

The external profiling measures the execution time of scanning text of 32 MB. Figure 5 presents the benchmark results with LSP = 8 first, where LSP denotes the length of the shortest pattern. The benchmark results for LSP < 8 will be

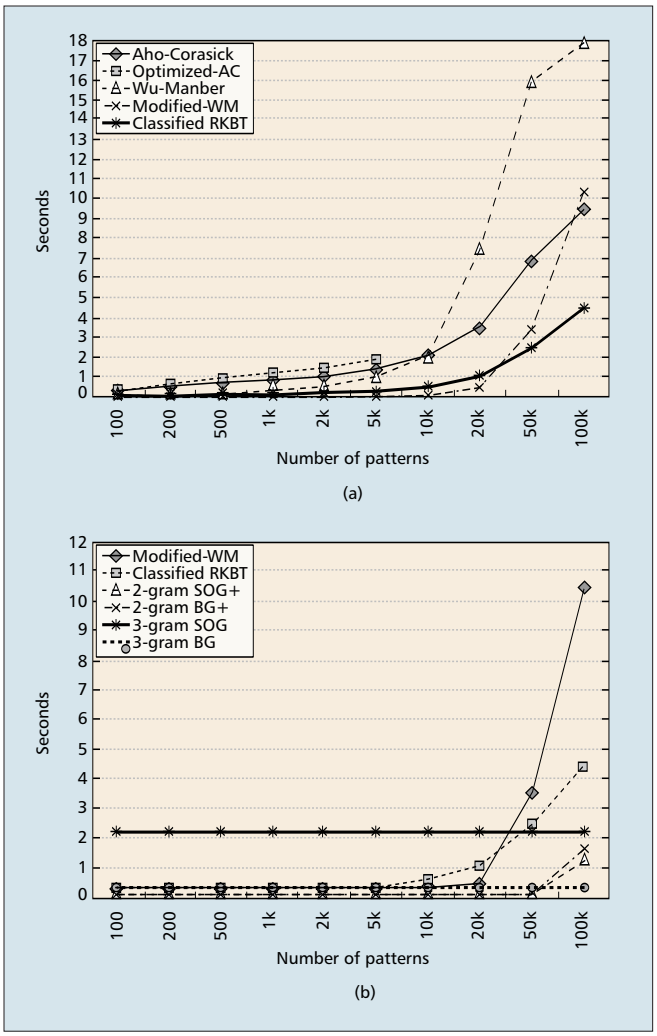

Figure 5. *Comparison of execution time of selected string matching algorithms.*

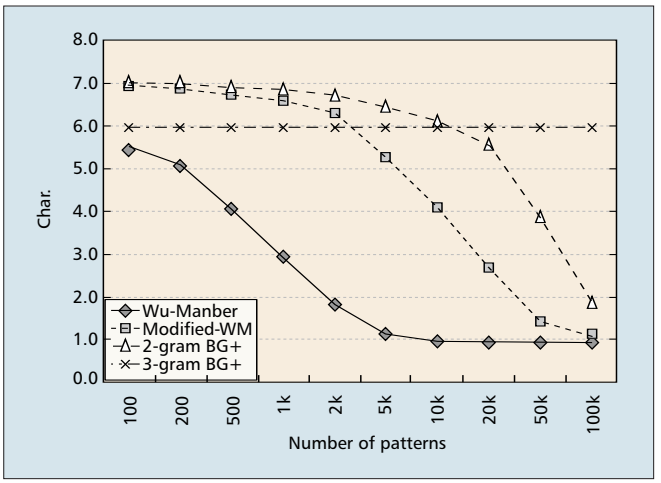

Figure 6. *Comparison of the profiling results of average shift distance.*

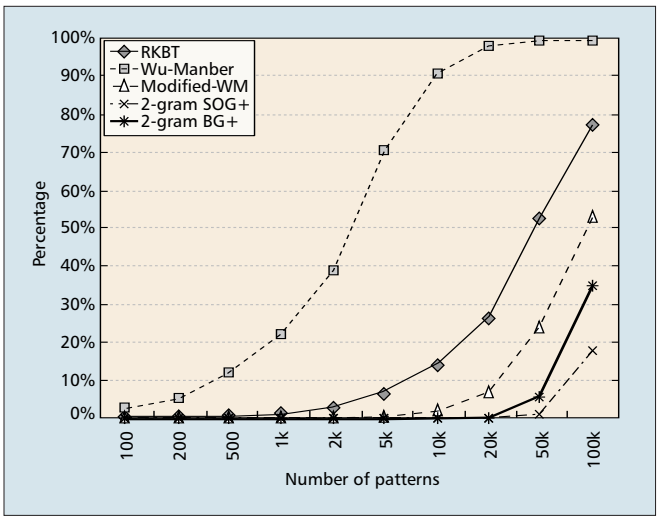

Figure 7. *Comparison of the percentage of possible matches for each algorithm.*

discussed later. To avoid many curves overlapping in one figure, the presentation of curves is separated into Fig. 5a and 5b for clarity.

Figure 5a demonstrates that the Modified-WM algorithm is the most efficient when the pattern set size is smaller than 20,000. When the pattern set size is greater than 20,000, the CRKBT algorithm is the most efficient. By the way, the execution time of the Optimized AC for the pattern set size larger than 5,000 is not presented because the execution takes too long to stop. The problem might be due to a bug in the original implementation of Snort. The execution time of the Modified-WM and CRKBT algorithms is compared with that of the BG + and SOG + algorithms in Fig. 5b. The 2-gram BG + algorithm is the fastest when the pattern set size is smaller than 50,000. For the pattern set size greater than 50,000, the 3-gram BG + algorithm is the fastest. This explanation is left to the internal profiling in the next section.

Internal Profiling

The internal profiling intends to justify the observations in the external profiling. For example, why is the Modified-WM algorithm more efficient than the WM algorithm? Why is the BG + algorithm very efficient? The effects of the average shift distance, the percentage of possible matches, and the number of memory accesses of each algorithm are profiled as follows.

Shift Distance — Both the WM and BG + algorithms allow skipping certain characters in the shift of the search window. This benchmark profiles the average shift distance of both algorithms. Figure 6 presents the profiling results. The average shift distance of the WM algorithm is close to one character when the pattern set size is between 5,000 and 100,000, so the WM algorithm can barely skip a character in this case. The average shift distance of the Modified-WM algorithm is greater than that of the WM algorithm, which can explain why the Modified-WM algorithm is more efficient than the WM algorithm. The results also explain why the 2-gram BG + algorithm is the fastest when the pattern set size is small, and why the 3-gram BG + algorithm is the fastest for a large pattern set due to the long average shift distance.

The Percentage of Possible Matches — Some algorithms filter the text first for possible matches and then verify whether a true match occurs. As the number of possible matches increases, the string matching will spend more time in verification and the verification can dominate the execution. The percentage of possible matches in the shifts is profiled for each algorithm.

Figure 7 shows the percentage of possible matches for each algorithm. The Modified-WM algorithm has fewer possible

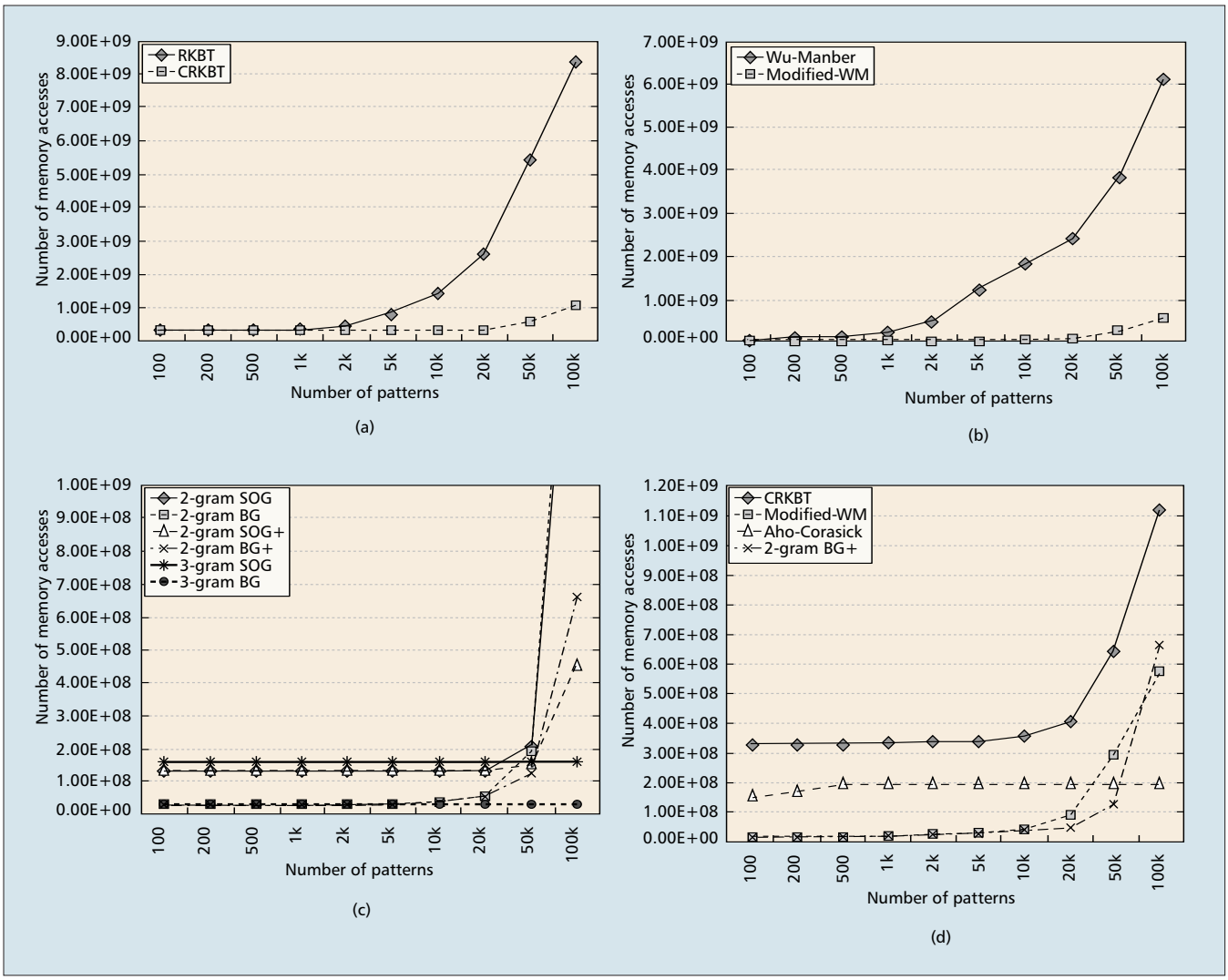

Figure 8. *Comparison of the number of memory accesses in each algorithm.*

matches than the WM algorithm. This also explains why the Modified-WM algorithm is faster. The fast increasing of possible matches in the WM algorithm indicates that the WM algorithm is not scalable to a large pattern set. This figure also explains why the BG + algorithm is faster than the Modified-WM algorithm because of its fewer possible matches.

Memory Accesses — It is insufficient to justify the external profiling results solely from the shift distance and the percentage of possible matches. For example, why is the CRKBT algorithm the fastest in Fig. 5a as the pattern set size is larger than 50,000? This section attempts to observe the reason from memory accesses.

Figure 8a–c each presents the number of memory accesses of the algorithms in the same category (categorized earlier) profiled by Valgrind [39]. For algorithms in the same category, the fewer the memory accesses, the faster the algorithm in the external profiling. However, this is not the case for algorithms in different categories, as presented in Fig. 8d. For instance, the CRKBT algorithm has more memory accesses than the AC algorithm, but the former is faster. The number of memory accesses is insufficient to justify the results in the external benchmarks.

More memory accesses do not imply longer time spent in accessing the memory. The number of cache misses can affect the performance significantly. We profile the number of cache misses to justify this point. Figure 9 shows the cache misses for the CRKBT algorithm are fewer than that for the Modified-WM and AC algorithms. The number of cache misses for the 2-gram BG + algorithm is the least. According to the cache misses, we can justify the prior results, including that the CRKBT algorithm is more efficient than the Modified-MW and AC algorithms for a large pattern set. In addition, the efficiency of the 2-gram BG + algorithm is also justified.

In addition to profiling the number of memory accesses, the memory consumption of each algorithm is also observed in Fig. 10. The CRKBT and 2-gram BG + algorithms have slow growth in memory consumption, which primarily comes from the increasing size of the ordered tables for binary search. The Modified-WM algorithm uses fixed size of memory for building the shift table and hash table for verification. The memory consumption in both the AC and Optimized-AC algorithms grow larger than that in the others as the pattern set increases. The memory consumption of the Optimized AC algorithm for the pattern set size larger than 5,000 is not presented because a possible bug impedes the correct execution.

Profiling for Short Patterns and Summary

The external and internal profiling demonstrates that the 2-gram BG + algorithm is the fastest for LSP = 8 with the pattern set size smaller than 50,000, and the 3-gram BG +

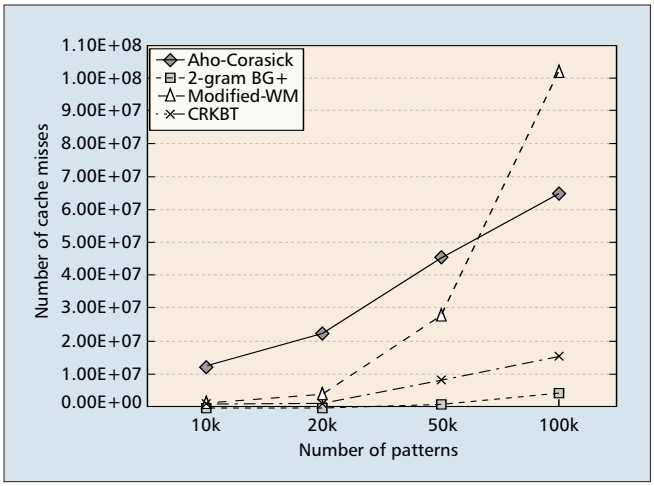

Figure 9. *Comparison of the number of cache misses in each algorithm.*

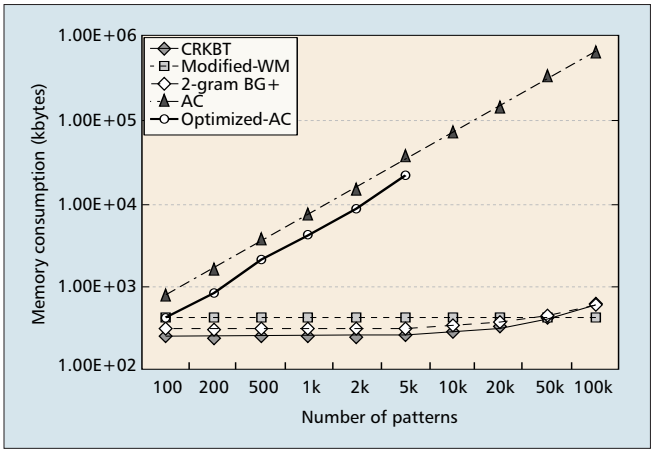

Figure 10. *Comparison of the memory consumption in each algorithm.*

algorithm is the fastest for LSP = 8 with a larger pattern set size. In addition, we also profile the performance for LSP between 1 and 7. The ranks of each algorithm in efficiency for LSP between 4 and 7 are similar to that for LSP = 8, so the benchmark results are not presented here. However, the ranks for LSP between 1 and 3 differ. Figure 11 shows that the AC, FNPw2, and Modified-WM algorithms are the fastest algorithm for LSP = 1, 2, and 3, respectively. Figure 12 summarizes the fastest algorithm for different pattern set sizes and pattern lengths.

Experiments on Real Applications

Implementation in Three Content Security Packages

Each network content security application has different pattern lengths and pattern set size, as discussed earlier. The shadow area in Fig. 13 indicates the range of pattern lengths and pattern set size of each application, overlapping with the profiling results in Fig. 12. The shaded arrows indicate an application has some patterns longer than the lengths in the range of the shadow area. This figure can suggest which algorithm to be better implemented in each application. We revise the original content security packages mentioned previously by implementing the suggested algorithms accordingly and observe the acceleration below.

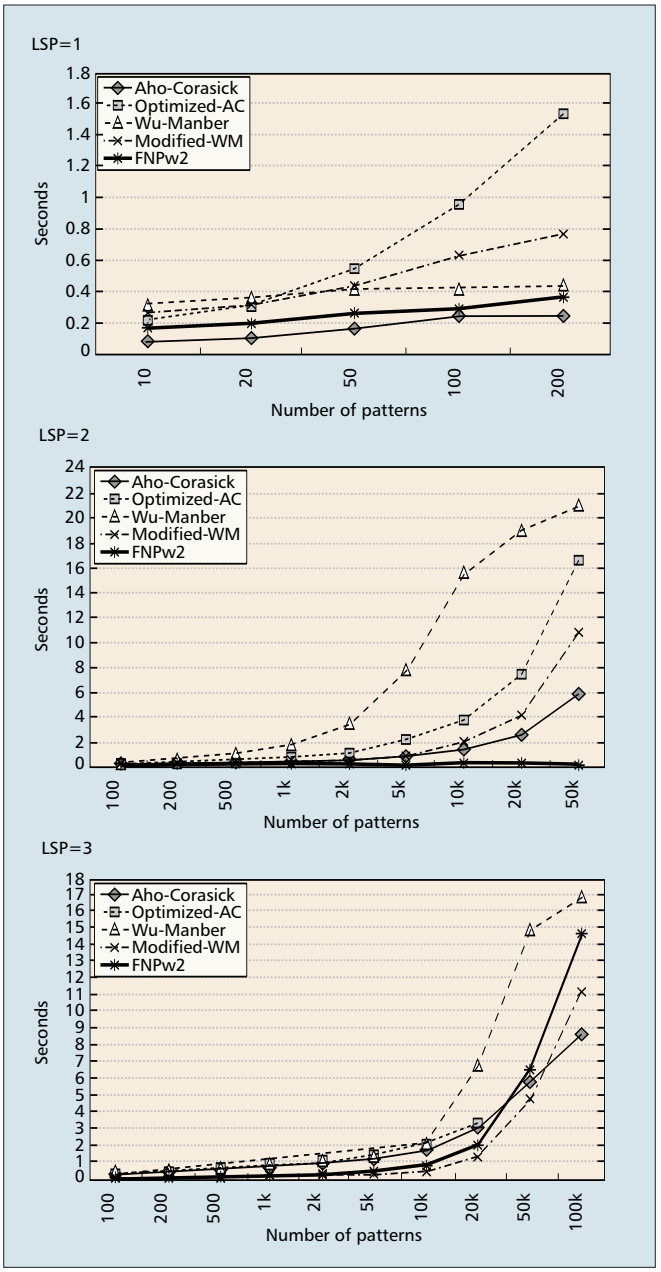

Figure 11. *Comparison of the execution time for LSP = 1, 2 and 3. (FNPw2 denotes FNP with w = 2.)*

ClamAV — The LSP of basic patterns in ClamAV is 10 and the pattern set size of them is larger than 30,000 to date. We replace the WM algorithm with the 2-gram BG + algorithm to handle exact matching of basic patterns described earlier. When the pattern set size is even larger in the future, the 3-gram BG + algorithm should be used to enhance the efficiency. In addition, the AC algorithm keeps matching regular expressions of multi-part patterns mentioned earlier.

DansGuardian — According to our investigation, 25 content keywords are scanned with the Horspool algorithm one by one in the current implementation. If the *forcequicksearch* option is enabled, every pattern in the pattern set will be searched for with the Horspool algorithm. Enabling this option will have the text scanned as many times as the number of patterns. We do not enable this option because multiple passes through the content will actually slow down the search. We group all the patterns together and implement the

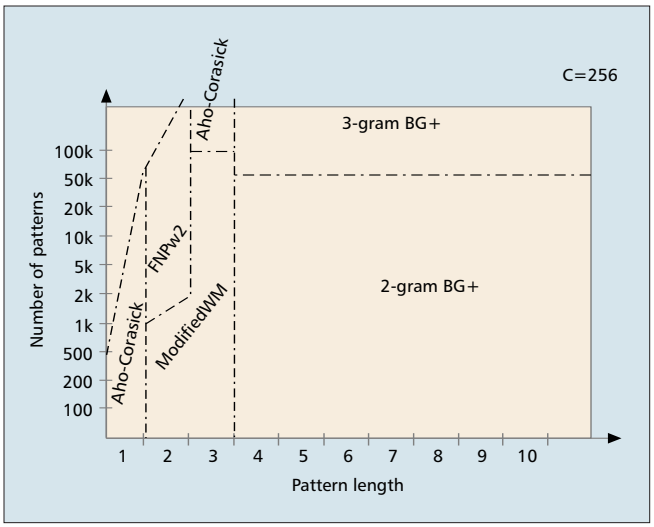

Figure 12. *The fastest algorithm for different pattern set sizes and pattern lengths. (C denotes |Σ|.)*

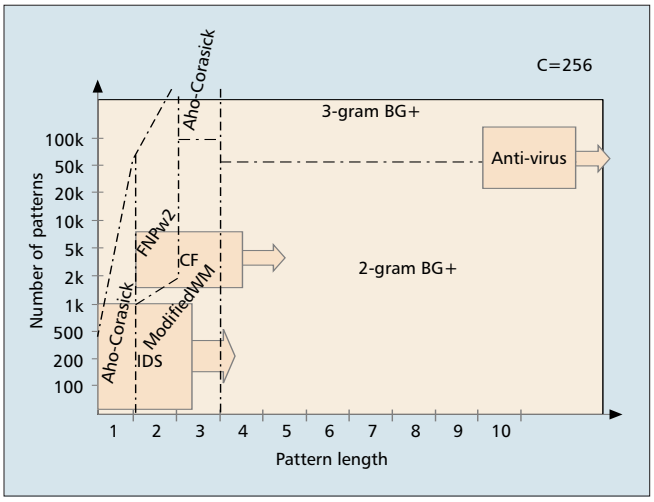

Figure 13. *The profiling summary. (C denotes |Σ|.)*

Modified-WM algorithm to handle short patterns with LSP = 2 and 3, and the 2-gram BG + algorithm to handle the longer patterns.

Snort — Snort groups patterns into rule sets according to the packet header. The LSP of every rule set is not the same. We implement a hybrid method instead of enabling the default method, the Modified-WM algorithm. The AC algorithm is selected for LSP = 1; otherwise, the Modified-WM algorithm still handles the pattern matching.

Benchmarking of the Revised Implementation

The speed-up of each revised package is benchmarked in this section. Besides, the performance for both the real and synthetic sample data is also compared. Here the synthetic data are generated from uniformly distributed random characters in the character set. The comparison of both types of sample data can exhibit whether the observation for the synthetic data is also applied to real situations.

Benchmarking for ClamAV — We select 10 Windows execution files having the sizes between 32 KB and 16 MB as real data in the benchmark. This benchmark also tests for synthetic data of the same size. The execution time of both the original ClamAV and its revised version is measured. Figure 14 exhibits the improvement of the revised implementation. As the file size increases, the difference in scanning time between both versions becomes obvious. For example, the revision is five times faster than the original when the file size is 16 MB. The acceleration comes primarily from that reduction of verification during the search. Figure 14 also compares the execution time for real and synthetic data in both versions. The difference between both types of data is almost unnoticeable because the character distribution in the patterns and files is close to random in ClamAV.

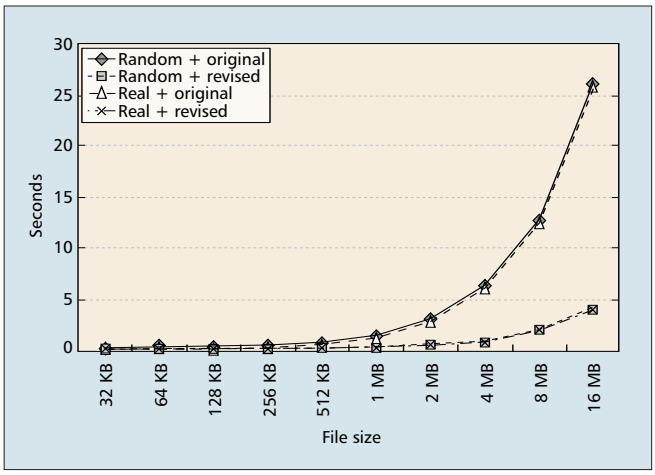

Figure 14. *The performance improvement for both random and real data in the revised version of ClamAV.*

Benchmarking for DansGuardian — We use *wget* (http://www.gnu.org/software/wget/wget.html) to mirror an RFC Web site at http://asg.web.cmu.edu/rfc/rfc-index.html that contains more than 8,000 files, including HTML files and ordinary text files. These files are scanned by DansGuardian's content filtering function. Figure 15 shows that the original implementation takes 2128 seconds to mirror the entire site, while the revised implementation needs 1708 seconds to finish. The acceleration is insignificant because DansGuardian needs to find out all content keywords. The verification algorithm has to look at every possible match that has the same hash value. The filtering part in the entire searching process becomes less significant, and so is its acceleration.

We also generate synthetic Web pages for comparison with the real Web pages. The sizes and names of these files are the same as all the RFC files except the content. First, we generate data from the character set of 256 characters and observe the difference between real data and synthetic data. The execution for synthetic data is faster than that for real data, because the character set distribution of synthetic data is close to uniform distribution, but that of real data is biased toward the English character set. Because the characters in real data concentrate more on English characters, the character set is effectively to be a small one. More possible matches occur and more verification is required than those for a uniformly distributed character set.

The character set of only 26 characters is also tested. The probability of possible matches increases, and the processing time of content inspection becomes three times longer than that in the prior experiment with a character set of 256 characters. However, the efficiency for the synthetic data from the character set of 26 characters is still much faster than that for the real data because keywords are more likely to appear in real data than in randomly generated synthetic data so that more possible matches occur and more verification is required.

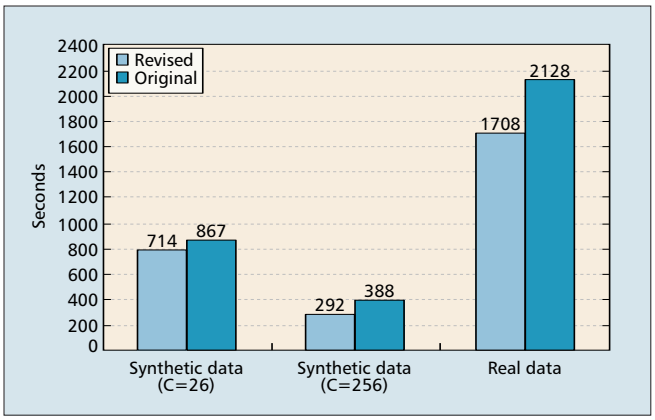

Figure 15. *The performance improvement for both random and real data in the revised version of DansGuardian. (C denotes $|\Sigma|$.)*

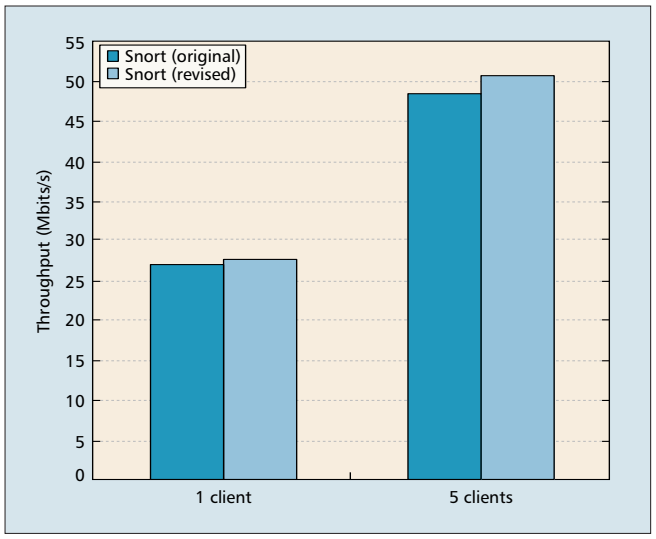

Figure 16. *The benchmarking result of Snort.*

Benchmarking for Snort — HTTP traffic accounts for a large quantity of the total traffic in the Internet, so we feed HTTP traffic to Snort as we did in the previous section for this benchmark. In addition, Snort is configured to run in the inline mode in which the traffic will pass through Snort for easy measurement of its throughput. Figure 16 presents the benchmarking results of the throughput. First, we use a single client to mirror the entire site. The acceleration of the revised Snort inspection is insignificant. We then add up to five clients, i.e., more traffic, and the acceleration of the revised version becomes a little more obvious. However, the enhancement is still insignificant as Snort only inspects the HTTP header instead of the HTTP header plus HTTP body in most cases [40]. Only a small portion of the traffic is inspected, so the acceleration is not that obvious.

The comparison between real data and synthetic data is not carried out here because the content inspection in Snort is mostly restricted within the HTTP header. If we generate random data instead of the real HTTP header, the transactions between the HTTP client and the HTTP server are unable to proceed, nor is the benchmark.

Conclusions

This research reviews and profiles some typical string matching algorithms to observe their performance under various conditions and gives an insight into choosing the most efficient algorithm for designing network content security applications. The AC algorithm is suitable for LSP = 1. The Modified-WM algorithm is suitable for LSP = 2 when the pattern set size is smaller than 1,000, and the FNPw2 algorithm is suitable for LSP = 2 when the pattern set size is larger than 1,000. The Modified-WM algorithm is suitable for LSP = 3, and the BG + algorithm is suitable for LSP ≥ 4. These results are also justified by means of implementing them in real applications. The new implementation is shown to improve the performance. These results will also help to select an efficient algorithm for future applications.

The CRKBT algorithm is proposed to accelerate the original RKBT algorithm up to four times faster for a pattern set of as large as 100,000 patterns. Moreover, the BG + and SOG + algorithms that use CRKBT as the verification algorithm are twice faster than the original algorithms.

This work also observes the difference between performance for the real and synthetic data on practical applications. The performance of ClamAV is insensitive to the synthetic data or the real data because the character set distribution is close to uniform. However, the performance of DansGuardian can be much different for the real data than for the synthetic data because both the patterns and text in DansGuardian are biased to the English vocabulary. Besides, DansGuardian may spend a long time in verifying possible matches because many content keywords have the same hash value and thus more comparisons are needed.

Acknowledgment

This work was supported in part by the Program of Excellence Research, National Science Council in Taiwan, and in part by grants from Cisco and Intel.

References

[1] P. Gupta and N. McKeown, "Algorithms for Packet Classification," *IEEE Network*, vol. 15, no. 2, Mar.–Apr. 2001, pp. 24–32.
[2] S. Antonatos, K. G. Anagnostakis, and E. P. Markatos, "Generating realistic Workloads for Network Intrusion," *Proc. ACM 4th Int'l. Wksp. Software and Performance (WOSP)*, Redwood, CA, Jan. 2004.
[3] S. Antonatos *et al.*, "Performance Analysis of Content Matching Intrusion Detection Systems," *Int'l. Symp. Applications and the Internet (SAINT'04)*, Tokyo, Japan, Jan. 2004.
[4] M. Fisk and G. Varghese, "Fast content-Based Packet Handling for Intrusion Detection," UCSD Tech. Rep. CS2001–0670, 2001.
[5] Y. D. Lin *et al.*, "Designing an Integrated Architecture for Network Content Security Gateways," *IEEE Computer*, May 2006.
[6] G. Navarro and M. Raffinot, *Flexible Pattern Matching in Strings*, Cambridge University Press, 2002.
[7] R. Muth and U. Manber, "Approximate Multiple String Search," Combinatorial Pattern Matching (CPM), Lecture Notes in Computer Science 1075, Laguna Beach, CA, 1996. pp. 75–86.
[8] J. Kytojoki, L. Salmela, and J. Tarhio, "Tuning String Matching for Huge Pattern Sets," CPM 2003, LNCS 2676, Morelia, Mexico, June 2003, pp. 211–24.
[9] M. Chrochemore and W. Rytter, *Jewels of Stringology*, World Scientific Publishing, 2003.
[10] I. Sourdis, "Efficient and High-Speed FPGA-Based String Matching For Packet Inspection," M.Sc. Thesis, Technical University of Crete, Greece, July 2004.
[11] SafeNet SafeXcel 4850, product information available: http:// http://www.safenet–inc.com/products/chips/safeXcel4850.asp
[12] Tarari RegEx Content Processor 4, Product information available: http://www.tarari.com/regexEAP/index.html
[13] G. Navarro, "A Guided Tour to Approximate String Matching," *ACM Computing Surveys*, vol. 33, no. 1, Mar. 2001, pp. 31–88.

[14] L. Cleophas, B. W. Watson and G. Zwaan, "A New Taxonomy of Sub–Linear Keyword Pattern Matching Algorithms," Department of Mathematics and Computer Science, Technische Universiteit Eindhoven, Apr. 22, 2004.
[15] G. Navarro and M. Raffinot, "Fast and Flexible String Matching By Combining Bit-Parallelism and Suffix Automata," *ACM J. Experimental Algorithms*, vol. 5, 2000, pp. 1–36.
[16] A. Aho and M. Corasick, "Efficient String Matching: An Aid to Bibliographic Search," *Commun. ACM*, vol. 18, no. 6, 1975, pp. 333–40.
[17] M. Norton, "Optimizing Pattern Matching for Intrusion Detection," Available: http://www.snort.org/docs/
[18] N. Tuck *et al.*, "Deterministic Memory–Efficient String Matching Algorithms for Intrusion Detection," *IEEE INFOCOM*, Hong Kong, Mar. 2004.
[19] L. Tan and T. Sherwood, "A High Throughput String Matching Architecture for Intrusion Detection and Prevention," *Int'l. Symp. Comp. Architecture (ISCA)*, Madison, WI, June 2005.
[20] R. Boyer and S. Moore, "A Fast String Searching Algorithm," *Commun. ACM*, vol. 20, no. 10, 1977, pp. 762–72.
[21] A. C. Yao, "The Complexity of Pattern Matching for a Random String," *SIAM J. Computing*, vol. 8, no. 3, 1979, pp. 368–87.
[22] Z. Galil, "On Improving the Worst Case Running Time of the Boyer Moore String Matching Algorithm," *Commun. ACM*, vol. 22, no. 9, 1979, pp. 505–08.
[23] N. Horspool, "Practical Fast Searching In Strings," *Software – Practice and Experience*, vol. 10, 1980, pp. 501–06.
[24] S. Wu and U. Manber, "A Fast Algorithm For Multi–Pattern Searching," Report TR–94–17, Department of Computer Science, University of Arizona, 1994.
[25] R. T. Liu *et al.*, "A Fast String–Matching Algorithm For Network Processor-Based Intrusion Detection System," *ACM Trans. Embedded Computing Systems*, vol. 3, no. 3, Aug. 2004, pp. 614–33.
[26] R. T. Liu *et al.*, "A Fast Pattern–Match Engine For Network Processor-Based Network Intrusion Detection System," *Proc. Information Technology: Coding and Computing (ITCC)*, vol. 1, Las Vegas, NV, Apr. 2004, pp. 97–101.
[27] R. Karp and M. Rabin, "Efficient Randomized Pattern-Matching Algorithms," *IBM J. Research and Development*, vol. 31, no. 2, Mar. 1987, pp. 249–60.
[28] R. Baeza-Yates and G. Gonnet, "A New Approach to Text Searching," *Commun. ACM,* vol. 35, 1992, pp. 74–82.
[29] G. Navarro and M. Raffinot, "A Bit-Parallel Approach to Suffix Automata: Fast Extended String Matching," *Proc. Combinatorial Pattern Matching (CPM)*, Piscataway, NJ, July 1998, pp. 14–33.
[30] C. Charras and T. Lecroq, Exact String Matching Algorithms, available http://www–igm.univ–mlv.fr/~lecroq/string
[31] E. P. Markatos *et al.*, "Exclusion-Based Signature Matching For Intrusion Detection," *Proc. IASTED Communications and Computer Networks (CCN)*, Cambridge, MA, Nov. 2002, pp. 146–52.
[32] K. G. Anagnostakis *et al.*, "E2XB: A Domain Specific String Matching Algorithm for Intrusion Detection," *Proc. 18th IFIP Int'l. Information Security Conference (SEC)*, Athens, Greece, May 2003.
[33] C. Allauzen and M. Raffinot, "Factor Oracle of a Set of Words," Tech. Rep. 99–11, Institute Gaspard-Monge, Université de Marne–la–Vallée, 1999.
[34] C. J. Coit, S. Staniford and J. McAlerney, "Towards Faster String Matching for Intrusion Detection or Exceeding the Speed of Snort," *Proc. 2nd DARPA Information Survivability Conf. and Exposition (DISCEX II)*, June 2001.
[35] Y. Miretskiy *et al.*, "Avfs: An On-Access Anti-Virus File System," *USENIX Security Symp.*, San Diego, CA, 2004.
[36] M. Norton and D. Roelker, "Multi–Rule Inspection Engine," available: http://www.snort.org/docs/
[37] Perl Compatible Regular Expressions (PCRE), available: http://www.pcre.org
[38] S. Wu and U. Manber, "Agrep – A Fast Approximate Pattern–Matching Tool," *Proc. USENIX Winter 1992 Tech. Conf.*, San Francisco, CA, 1992, pp. 153–62.
[39] Valgrind, available http://valgrind.org/
[40] M. Norton and D. Roelker, "Snort 2.0 protocol Flow Analyzer," available: http://www.snort.org/docs/

BIOGRAPHIES

PO-CHING LIN (pclin@cis.nctu.edu.tw) is a Ph.D. candidate in the Department of Computer Science at National Chiao Tung University. His research interests include network security, string-matching algorithms, hardware software codesign, content networking, and performance evaluation. He received an MS degree in computer science from National Chiao Tung University.

ZHI-XIANG LI (lizx@cis.nctu.edu.tw) is an MS student in the Department of Computer Science at National Chiao Tung University. His research interests include string-matching algorithms, network security, content networking, high-speed networking, and performance evaluation. He received a BS degree in computer science from National Chiao Tung University.

YING-DAR LIN (ydlin@cis.nctu.edu.tw) is a professor in the Department of Computer Science at National Chiao Tung University, Hsinchu, Taiwan. His research interests include design, analysis, implementation, and benchmarking of network protocols and algorithms; wire-speed switching and routing; and embedded hardware software co-design. He received a Ph.D. in computer science from the University of California, Los Angeles. He is a member of the IEEE and the ACM.

YUAN-CHENG LAI (laiyc@cs.ntust.edu.tw) is an associate professor in the Department of Information Management at National Taiwan University of Science and Technology, Taipei, Taiwan. His research interests include high-speed networking, wireless network and network performance evaluation, Internet applications, and content networking. He received a Ph.D. in computer science from National Chiao Tung University.

FRANK C. LIN (fclin@cisco.com.) is a technical lead of Cisco Systems Inc. His current interest is in automatic vulnerability inspection and security testing. He has been in the computer, networking, and telecommunication industry for 30 years, serving at Calcomp in Taiwan, Univac/Sperry/Unisys in Salt Lake City, Octel, and Ardent/Cisco in Silicon Valley. He received his Ph.D. in CS from the University of Utah in 1986.