Realizing a Sub-Linear Time String-Matching Algorithm With a Hardware Accelerator Using Bloom Filters

Po-Ching Lin, *Member, IEEE*, Yin-Dar Lin, *Senior Member, IEEE*, Yuan-Cheng Lai, Yi-Jun Zheng, and Tsern-Huei Lee, *Senior Member, IEEE*

***Abstract*—Many network security applications rely on string matching to detect intrusions, viruses, spam, and so on. Since software implementation may not keep pace with the high-speed demand, turning to hardware-based solutions becomes promising. This work presents an innovative architecture to realize string matching in sub-linear time based on algorithmic heuristics, which come from parallel queries to a set of space-efficient Bloom filters. The algorithm allows skipping characters not in a match in the text, and in turn simultaneously inspect multiple characters in effect. The techniques to reduce the impact of certain bad situations on performance are also proposed: the *bad-block* heuristic, a linear worst-case time method and a non-blocking interface to hand over the verification job to a verification module. This architecture is simulated with both behavior simulation in C and timing simulation in HDL for antivirus applications. The simulation shows that the throughput of scanning Windows executable files for more than 10 000 virus signatures can achieve 5.64 Gb/s, while the worst-case performance is 1.2 Gb/s if the signatures are properly specified.**

***Index Terms*—Algorithms, field-programmable gate arrays (FPGAs), string matching.**

I. Introduction

DEEP content inspection at the application layer to detect proliferating intrusions and viruses on the Internet is a known critical part to the performance. The inspection involves string matching for multiple patterns of malicious signatures. Numerous algorithms have been developed for efficiency over the past decades [1]. Software implementation of string-matching algorithms to handle the increasing Internet traffic becomes more challenging than ever. Extensive study thus turns to specialized hardware accelerators to meet the high-speed demand on the order of multi-gigabits per second.

Manuscript received November 23, 2006; revised July 06, 2007. First published April 24, 2009; current version published July 22, 2009. This work was supported in part by the Taiwan National Science Council's Program of Excellence in Research and by grants from Cisco and Intel.

P. C. Lin, Y. D. Lin, and Y. J. Zheng are with the Department of Computer Science, National Chiao Tung University, Hsinchu 300, Taiwan (e-mail: pclin@cis.nctu.edu.tw; ydlin@cs.nctu.edu.tw; yjzheng@cis.nctu.edu.tw).

Y. C. Lai is with the Department of Information Management, National Taiwan University of Science and Technology, Taipei 106, Taiwan (e-mail: laiyc@cs.ntust.edu.tw).

T. H. Lee is with the Department of Communication Engineering, National Chiao Tung University, Hsinchu 300, Taiwan (e-mail: tlee@banyan.cm.nctu.edu.tw).

Digital Object Identifier 10.1109/TVLSI.2008.2012011

Many FPGA accelerators hardwire signatures into logic cells [2], [3], and match several characters in the text per cycle with pipelining for high throughput. However, the gate count constrains the number of signatures that can be hardwired. Frequent dynamic signature updating is also costly due to long reprogramming time. Implementing the designs in application-specific integrated circuit (ASIC) is infeasible since an ASIC chip is not reconfigurable. Storing signatures in the memory simplifies the updating [4], [5]. The size of external memory on the order of GBs also increases the scalability of the number of signatures.

A common memory-based approach sequentially reads each character in the text to track a finite automaton that accepts the patterns in the pattern set, so its time complexity is linear [6]. Some designs can accelerate the process by tracking multiple characters at once [7]–[9], but the hardware or space complexity also increases with the number of characters under tracking. Another approach moves a search window through the text to check whether it contains a suspicious match or not [10]–[12]. Assuming most of the data is legitimate, this approach can quickly exclude the legitimate data, and verifies only the suspicious matches. The window is generally advanced by only one character at once for not missing any possible match. Duplicating multiple copies of hardware engines can advance the window faster, but the degree of parallelism is subject to availability of hardware resources.

A class of algorithms can *skip* characters not in a match based on algorithmic heuristics to inspect multiple characters at once in effect, and have been widely implemented in practical software [13]. This work borrows the idea of algorithmic heuristics, instead of sheer relying on duplicating hardware engines or high operating frequency. These algorithms are rarely realized in hardware so far, perhaps due to the following two reasons.

1) Calculating algorithmic heuristics involves looking up a large table, which may not fit in the embedded memory, but accessing the table in the external memory is slow.
2) The worst-case performance of such algorithms may be worse than that of linear-time algorithms.

Such algorithms are less resilient to some bad cases, such as nonuniform character distribution that shortens the skipping distance and algorithmic attacks that attempt to exploit the worst case. Despite the drawbacks, we believe sub-linear time algorithms deserve the study since they are generally fast and do not rely on massive hardware parallelism for their speed.

We propose an innovative architecture to realize a sub-linear time algorithm, namely the bloom filter accelerated sub-linear

1063-8210/$26.00 © 2009 IEEE

TABLE I
IMPORTANT NOTATIONS THROUGHOUT THIS PAPER

notation	*description*
$\mathscr{P}$	The pattern set.
$\mathscr{P}'$	The set of pattern prefixes under consideration during preprocessing and scanning.
P_i	The i-th pattern in the pattern set.
$P_i[j \dots k]$	A substring from the j-th to the k-th character of P_i.
Σ	The character set. $\|\Sigma\| = 256$ in this paper.
r	The number of patterns in the pattern set.
n	The text length.
m	The shortest pattern length in the pattern set. Also the length of the search window.
b	The block size. $b = 4$ in this paper.
s	The shift value.
v	The size of the bit vector in a Bloom filter.
$BF(G_j)$	The Bloom filter storing the group G_j.
B_j	The block that is j characters backward away from the last character in the search window.

time (BFAST) algorithm. This architecture uses a set of Bloom filters [14], each representing a group of strings in a space-efficient bit vector for membership query. The algorithmic heuristics are derived from simultaneous queries to the Bloom filters to determine the shift distance of the search window. A suspicious match is handed over to a verification engine for verification without blocking the scan. Pipelining is also implemented to further increase the throughput by four times. A heuristic similar to the bad-character heuristic in the Boyer–Moore algorithm [15], namely the *bad-block* heuristic, can reduce the verification frequency and exploit larger shift values. A linear worst-case time option is also proposed to guarantee the time complexity.

The rest of this paper is organized as follows. Section II reviews typical string matching algorithms and hardware accelerators. Section III presents the architecture of the BFAST algorithm. Section IV presents the detailed hardware implementation. Section V evaluates this architecture and compares it with existing works. Section VI concludes this work.

II. Existing Works and Literature Background

A multiple-string matching algorithm searches the text $\mathsf{T} = t_1 t_2 \cdots t_n$ for occurrences of the patterns in a pattern set $\mathsf{P} = \{P_1, P_2, \dots, P_r\}$ on the same alphabet Σ, where r is the number of patterns. We use m to denote the shortest pattern length and assume $|\Sigma| = 256$ (number of values in a byte). Table I summarizes the notations in this paper.

A. String Matching Algorithms

The Aho–Corasick (AC) algorithm [6] feeds a finite automaton that accepts the patterns in the pattern set with the input characters one by one, so its time complexity is $O(n)$. A match is claimed if one of the final states is reached. Such automaton-based approaches, either deterministic finite automaton (DFA) or non-deterministic finite automaton (NFA), are common due to their flexibility in representing the patterns , and deterministic execution time for robustness to algorithmic attacks. The transition table of an automaton is compressed to reduce the memory requirement [16], [17]. Given the wide data bus in modern architectures, tracking one character at a time is inefficient. Several designs can determine the next state after reading a block of characters to boost the performance [7], [8], but they have two drawbacks: 1) compressing the transition table may need tricky techniques, if feasible, as the table grows with a large block and 2) because a signature may not start from a block boundary, the match engine should be duplicated several copies at the offset of one more character from the block boundary [8].

The Boyer–Moore (BM) algorithm is the first that can skip characters not in a match based on algorithmic heuristics [15], which are illustrated in [18]. Among the heuristics of the BM algorithm and its derivatives, we specifically mention the *bad-character* heuristic for its relevance to our work. This heuristic matches the characters backward from the suffix of the search window one by one, until either a mismatched character is found or the entire pattern is matched. If a mismatched character is found, the heuristic looks up a table to decide the shift distance of the window according to whether the character is in the pattern or not, and its position. However, the heuristic will significantly decrease the shift distance for a large pattern set due to the high probability of a character appearing in one of the patterns.

The WM algorithm matches a block of characters instead of a character to greatly reduce the chances that a block appears in the patterns. The algorithm assumes equal pattern lengths. If not, it considers only the first m characters of each pattern during preprocessing and scanning. The search window of m characters slides along the text during scanning according to the heuristics: if the right-most block of b characters in the search window appears in none of the patterns, a window shift by a maximum of $m - b + 1$ characters is safe without missing any match; otherwise, the shift value is $m - j$, where the right-most occurrence of the block in the patterns ends at position j. If the shift value is 0, i.e., the block is the suffix of some pattern, the occurrence of a true match is verified. The algorithm builds a shift table that keeps the shift values for indexing by the right-most block. Different blocks may be mapped to the same table entry, in which the minimum shift value of them is filled. This mapping saves the table space at the cost of smaller shift values. The worst performance of the WM algorithm may be poor. For example, if a pattern is aaaaa and the text is all a's, the search window cannot skip any character. The time complexity is $O(mn)$ because the verification takes $O(m)$ in every text position. Nonetheless, variants of the algorithm can be found in popular software, such as ClamAV (www.clamav.net) for anti-virus.

A Bloom filter compactly stores the patterns in a v-bit bit vector for membership queries [14]. For each pattern X, the filter sets to 1 the bits addressed by the k hash values $h_1(X),\ h_2(X), \dots, h_k(X)$ ranging from 0 to $v - 1$. When a substring W in the text is matched, a membership query looks up the bits addressed by W's hash values. If one of the bits is unset, W must not be in the pattern set; otherwise, verification follows to see whether a true match occurs. The uncertainty comes from different patterns setting checked bits. Properly choosing v and k can control the false-positive rate.

B. Hardware Accelerators

String-matching hardware accelerators either hardwire the patterns into logic cells on field-programmable gate array (FPGA) or store them in memory. Updating the patterns in the former may take hours to regenerate a bit-stream and a few minutes to download it onto the chip. Partial reconfiguration

can reduce the cost [19]. Besides the reconfiguration cost, the number of available gate counts limits the size of the pattern set. Several examples use this approach. For example, four scanning modules run in parallel to scan multiple packets concurrently in [20], and the throughput is up to 1.184 Gb/s. Cho *et al.* designed a pipelining architecture of discrete comparators [2]. A pattern match unit involves four sets of four 8-bit comparators to directly compare four consecutive characters in each stage. The matching results from each stage are fed to the next in the pipelining. The design was later enhanced by fully pipelining the entire system [21], and the throughput can be up to 11 Gb/s at 344 MHz, but its area cost is still high. Several following studies were devoted to area reduction, such as [3].

Reconfiguration in memory-based accelerators involves only updating the memory content, and the logics either remain intact or experience only a slight change. The designs may utilize an AC-style automaton [22]–[27], a filtering search window [10]–[12], [28], or both [8]. Whatever approach they take, a fundamental issue is that if the scanning proceeds by only one character at once, it demands high operating frequency for high speed. Some of them can advance several characters at once by multiple parallel engines, but the available hardware resources restrict the degree of parallelism.

III. BFAST ARCHITECTURE

A. Drawbacks of Using a Shift Table

The block size in the WM algorithm is critical to the performance for a large pattern set. Given r patterns, the verification probability is $1-(1-1/|\Sigma|^b)^r$, i.e., the probability that the right-most block is a pattern suffix. Increasing b can reduce the probability, but also demands a larger shift table (e.g., 256^3 entries in an uncompressed table for $b=3$). A block size larger than three is impractical due to the huge table size. Mapping multiple blocks to the same entry filled with the minimum shift value of these blocks can compress the table, but the compression reduces the shift values and increases the verification frequency [29].

The shift table keeps little information about the patterns but the shift values. The information such as whether a block appears in a specific position or appears multiple times in the patterns is lost, but the information is important to exploit larger shift distance. Moreover, if a shift value is zero, nothing can be done but moving the search window by one character after verification. We therefore abort using a shift table.

B. Deriving Shift Distance Using Bloom Filters

The BFAST algorithm enhances the heuristics from the WM algorithm. Let $P_i[j\cdots k]$ denote a substring from the jth character to the kth character of P_i. We define a function

$$\mathrm{Pre}_\tau(P_i)=\begin{cases}P_i[1\cdots\tau], & \text{if } \tau<|P_i|,\\ P_i & \text{otherwise.}\end{cases} \quad (1)$$

The BFAST algorithm searches for patterns in P' during scanning, where P' is the set of $\mathrm{Pre}_m(P_i)$ if $m\geq b$, or the set of $\mathrm{Pre}_b(P_i)$ otherwise. If any pattern in P' is found, whether a true match in P occurs is verified. Let B_0 be the rightmost block in the search window. The heuristic for B_0 is described as follows.

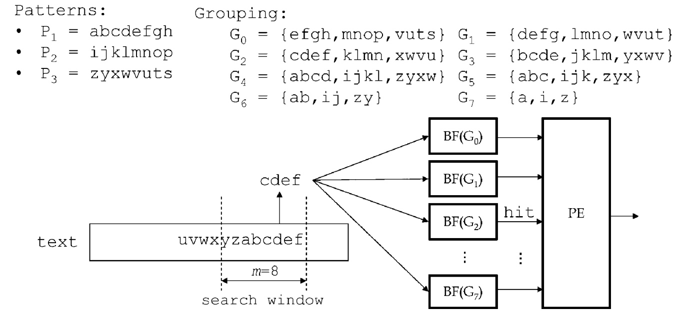


Fig. 1. Blocks in the patterns are grouped for deriving the shift value from querying Bloom filters in parallel. If a block is a member of some group G_j, $BF(G_j)$ must report a hit. The priority encoder (PE) determines the shift value.

1) If neither B_0 appears in the patterns nor any suffix of B_0 is a prefix of some pattern, the shift value is m if $m\geq b$, or b otherwise.
2) If B_0 does not appear in the patterns, but it has a suffix that is also the prefix of some pattern. Let k be the longest length of such a suffix. The shift value is $m-k$ if $m\geq b$, or $b-k$ otherwise.[1]
3) B_0 is a substring of some pattern if $m\geq b$, or a pattern is a substring of B_0. In the former, if the right-most occurrence of B_0 ends at position j of some pattern, the shift value is $m-j$. The *bad-block* heuristic depicted in Section III-C then evaluates whether additional checks are worthwhile to exploit a larger shift value. In the latter, a match is claimed directly.

This heuristic considers not only B_0 but also its suffix so that the maximum shift value can be m rather than $m-b+1$. Patterns shorter than b characters can be also handled.

The BFAST algorithm groups blocks in the patterns by their positions, so we can derive the position of every block in the search window by checking its membership in the groups. This method retains more information than a shift table, so it can use versatile heuristics. The shift value is derived by membership queries to a set of parallel Bloom filters, each of which stores an individual group.

Fig. 1 illustrates how to derive the shift value for $b=4$ in a trivial example. The blocks in the pattern set $\{P_1,P_2,P_3\}$ are grouped by position: G_0 is {efgh,mnop,vuts}, G_1 is {defg,lmno,wvut}, and so on. Let $BF(G_j)$ denote the Bloom filter storing G_j. These Bloom filters are queried in parallel for the membership of $B_0=\mathrm{cdef}$. Because cdef is a member of G_2, $BF(G_2)$ must report a hit. If no false positives occur in $BF(G_1)$ or $BF(G_0)$, the shift value is 2 according to the aforementioned heuristic. If none of the Bloom filters report a hit (i.e., neither B_0 appears in the patterns nor any suffix of B_0 is the prefix of some pattern), the maximum shift of $m=8$ characters is safe.

In general, when $m\geq b$, the groups are defined by (2) shown at the bottom of the next page. The queries check in parallel whether B_0 is a member of $G_0\cdots G_{m-b}$ and whether the k-character suffix of B_0 is a member of G_{m-k}, for

[1] We noticed Liu *et al.* [30] had a similar observation, but their heuristic based on the *prefix* rather than the *suffix* of the search window may skip over and overlook a suspicious match.

$k = 1 \cdots b - 1$. Because G_{m-1} and G_{m-2} contain only one or two characters, the Bloom filters of both are implemented as directly mapped tables for simplicity. When $m < b$, the groups are defined by (3) shown at the bottom of the page. The patterns in G_0 are further divided into $G_0^{(1)} \cdots G_0^{(b)}$, where $G_0^{(l)}$ are patterns of l characters. The queries check whether each substring of l characters in B_0 is in $G_0^{(l)}$ in parallel and whether the k-character suffix of B_0 is a member of G_{m-k}.

More than one Bloom filter may report a hit if a block makes multiple appearances in the patterns or false positives occur. A priority encoder can arbitrate and determine the shift value s as follows. If at least one Bloom filter reports a hit, $s = \min\{j|\ BF(G_j) \text{ reports a hit}\}$; otherwise, $s = m$. If no false positives occur, s is equal to that from the aforementioned heuristic because the Bloom filters report the exact membership of B_0. Otherwise, the false positive reported from a Bloom filter may make the shift shorter than it should be, but it is still safe—no match will be missed.

C. Bad-Block Heuristic in the Search Window

In practice, some blocks may appear much more frequently than the others. In the Windows executable files under our investigation, for example, the most frequent block "$0 \times 000 \times 000 \times 000 \times 00$" alone occupies 4.46% of the total blocks. If the suffix of some $P_i \in \mathsf{P}'$ happens to be a frequent block, the verification will be also frequent, following immediately after a hit in $BF(G_0)$. A verification failure also tells nothing but shifts the search window by only one character.

The BFAST algorithm avoids immediate verification by checking additional blocks $B_1, B_2, \ldots, B_{m-b}$ to exploit a larger shift value if needed, where B_j denotes the block that is j characters away from the last character backward in the search window. A heuristic similar to the bad-character heuristic, namely the *bad-block* heuristic, is described as follows.

1) Let H be $\{i|BF(G_i) \text{ reports a hit and } i \geq j\}$.
   a) If $\mathsf{H} \neq \emptyset$, the shift value derived by checking B_j is $i' - j$, where i' is the smallest value in H.
   b) Otherwise, the shift value is $m - j$. In (1), if $i' = j$, more checks may be needed as described in the following.

Theorem 1: The shift value derived here is safe.

Proof: Suppose a match occurs when the search window is shifted by a shorter distance. This means that either B_j or a suffix of B_j must be in one of the groups from $G_{i'-1}$ to G_j (in the first rule) or from G_{m-1} to G_j (in the second rule), implying that a Bloom filter between $BF(G_{i'-1})$ and $BF(G_j)$ or between $BF(G_{m-1})$ and $BF(G_j)$ will report a hit. This contradicts either that i' is the smallest such that $BF(G_{i'})$ reports a hit or that none of $BF(G_i)$ report a match for $i \geq j$. Therefore, the shift value will not miss a match. The heuristic in Section III-B is a special case for $j = 0$. ■

Implementing the rules is simple. The priority encoder just ignores the report from $BF(G_0) \cdots BF(G_{j-1})$ when B_j is checked. False positives in the Bloom filters may occur, but like the query from B_0, the shift is just shorter, but is still safe.

1) After B_j has been checked, where $j < m - b$, whether B_{j+1} should be checked next is evaluated based on the cost of additional checks to exploit a larger shift value. Let $\max_j(s)$ be the largest shift value derived since B_0 was checked, and $E_{j+1}[s]$ be the expected shift value when B_{j+1} is checked. The criterion

$$\frac{\lfloor E_{j+1}[s] \rfloor}{j+2} > \frac{\max_j(s)}{j+1} \qquad \max_j(s) = 0 \tag{4}$$

is evaluated (with integer division) to see if checking B_{j+1} is worthwhile. If the criterion is true, B_{j+1} will be checked next; otherwise, the search window will be moved by $\max_j(s)$ characters.

The estimate of $E_{j+1}[s]$ is precomputed for each j according to the analysis in Section III-F, and $\max_j(s)$ is updated after each block is checked. Because every shift value from B_0 to B_j is safe, $\max_j(s)$ is surely safe. If every Bloom filter from $BF(G_0)$ to $BF(G_j)$ reports a hit, a match may occur, and the checks should go on. In this case, $\max_j(s) = 0$ because the shift values derived from B_0 to B_j are all zeros. The equation $\max_j(s) = 0$ ensures the checks will continue. Only the inequality on the left-hand side is insufficient because $\lfloor E_{j+1}[s] \rfloor / j + 2$ might be zero due to integer division, even if $\lfloor E_{j+1}[s] \rfloor > 0$. The inequality may fail even though $\max_j(s) = 0$.

2) The verification procedure is invoked only if every block from B_0 to B_{m-b} gives rise to a hit in $BF(G_0), \ldots, BF(G_{m-b})$, respectively. The verification probability becomes only $\prod_{j=0}^{m-b} p_j$ for $m \geq b$, where p_j is the probability that $BF(G_j)$ reports a hit for B_j. The probability is normally low, particularly for long patterns such as virus signatures.

$$G_j = \begin{cases} \{P_i[m - j - b + 1 \cdots m - j]\ |P_i \in \mathrm{P}'\}, & \text{if } 0 \leq j \leq m - b, \\ \{P_i[1 \cdots m - j]\ |P_i \in \mathrm{P}'\}, & \text{if } m - b + 1 \leq j \leq m - 1 \end{cases} \tag{2}$$

$$G_j = \begin{cases} \mathsf{P}', & \text{if } j = 0, \\ \{\mathrm{Pre}_{b-j}(P_i)|P_i \in \mathsf{P}' \text{ and } |\mathsf{P_i}| > \mathsf{b - j}\}, & \text{if } 1 \leq j \leq b - 1. \end{cases} \tag{3}$$

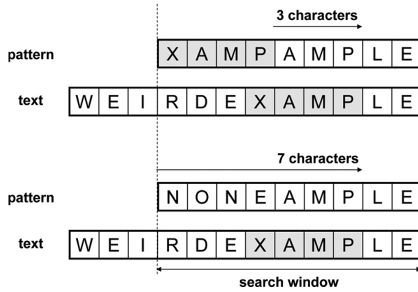


Fig. 2. Illustration of the bad-block heuristic.

Fig. 2 illustrates the bad-block heuristic with two trivial examples of only one pattern. In the upper example, we find MPLE in G_0, AMPL in G_1, but XAMP is in G_5. Then the shift distance of $5 - 2 = 3$ characters is safe. In the lower example, because neither XAMP nor its suffixes are in the groups from G_2 to G_8, the shift distance can be $9 - 2 = 7$ characters.

D. Performance in the Worst Case

The worst time complexity is $O(mn)$, when every block in the search window must be queried after each shift by one character. Manipulating to the worst case is not always feasible in practice. For example, an attacker knows a signature *malicious* and generates a string *nalicious* in the search window to force backward checks throughout the entire window. After the verification and the shift by one character, the right-most block in the next window becomes *ous*α, where $\alpha \in \Sigma$. The next shift distance will be at least $m - 1 = 8$ characters according to the heuristic in Section III-B. The manipulation fails, even though a series of strings nalicious are in the text. Galil discussed the general condition leading to the worst case in terms of the *periodicity* of a pattern [31]. Shortly put, properly specifying a signature to avoid a short period (i.e., it is not a prefix of u^i for $i > 1$, where u is a short string called a period) can reduce the risk of an algorithmic attack, as demonstrated above.

Several methods can guarantee the linear worst-case time complexity for a single fixed string or regular expression [31], [32], but none of them can guarantee so for multiple strings as far as we know. We thus suggest two alternatives to handle this problem. First, the worst case of $O(mn)$ is easily detected by calculating the average number of blocks that have been checked in the last η characters from the current text position, say $\eta = 100$. (A block may be counted more than once if it is revisited.) If the average is higher than a threshold, say $m\eta/2$, which is unusual in normal traffic, the available bandwidth of that flow is constrained to avoid a possible algorithmic attack. The approach has low cost, but may not work well when the attacks are from multiple flows.

Second, the worst time complexity is resulted from revisiting the blocks in the text many times during scanning. In Section V-A of [32], an approach of forward and backward scanning can assure no blocks are revisited in either direction to guarantee the linear time. This approach seems tantalizing, but its space complexity exponential to the number of characters in the patterns is prohibitively high for a large pattern set. We propose to borrow its concept of forward and backward scanning without revisiting in either direction, and use an assisting Aho–Corasick automaton instead of its original data structure. The procedure is described as follows.

1) The BFAST architecture searches the text until a suspicious match is found. Let the first character of the search window be the *critical position*.
2) The AC automaton tracks forward the characters from the critical position until the end of the window. The tracking will either: a) find the longest prefix of some pattern in the window suffix or b) go back to the initial state of the automaton (if a failure occurs). In case a), if the entire window is the pattern prefix, the tracking should be resumed beyond the window until the entire pattern is matched or a failure occurs somewhere. In both cases, the current automaton state is recorded, and the current text position plus one becomes the new critical position.
3) The search window is aligned with the prefix found in step 2) (i.e., their first characters are aligned), or to begin at the position where the failure occurs.
4) The BFAST architecture resumes its backward scanning in the new window. Two possibilities may occur.
    a) The backward scanning reaches the critical position [see Fig. 3(a)]. The procedure then goes back to step 2), in which the AC automaton resumes forward scanning from the critical position with the recorded state.
    b) The backward scanning gets a shift value from the heuristics before reaching the critical position, and the search window is shifted accordingly [See Fig. 3(b)]. The AC automaton then tracks the overlapping part of the new window and the last window. After the tracking, the current automaton state is recorded, and the current text position plus one becomes the new critical position. The procedure then goes back step 4).

Theorem 2: The procedure is correct, and its time complexity is linear in the worst case.

Proof: ***Correctness.*** The search window is shifted according to either forward scanning in Fig. 3(a) or bad-block heuristics in Fig. 3(b). In the former, if a pattern starts within the search window, its prefix must be in the window suffix, and the AC tracking in step 2) will find it. After the search window is shifted in step 3), the backward scanning in step 4) will reach the critical position because the search window is now aligned with the pattern. The AC tracking will be resumed from the critical position and eventually match the entire pattern. The shift thus will not miss a match. In the latter, we proved the shift will not miss a match in Section III-C.

Linear time. In each shift, the forward scanning is resumed either from or after the critical position where it is stopped last time (see Fig. 3), so the scanning never revisits the blocks in the text. Note that this algorithm looks for a *non-overlapping match*, which is sufficient for most network security applications [33]. When a match is found, the forward scanning will not revisit the blocks inside the match. Similarly, the backward scanning traverses before or until reaching the critical point, which is behind the end of the last window, so the backward scanning never revisits the blocks in the last window. Since neither direction revisits the blocks in the text, the blocks are read at most $2n$ times, and the worst time complexity is $O(n)$. ■

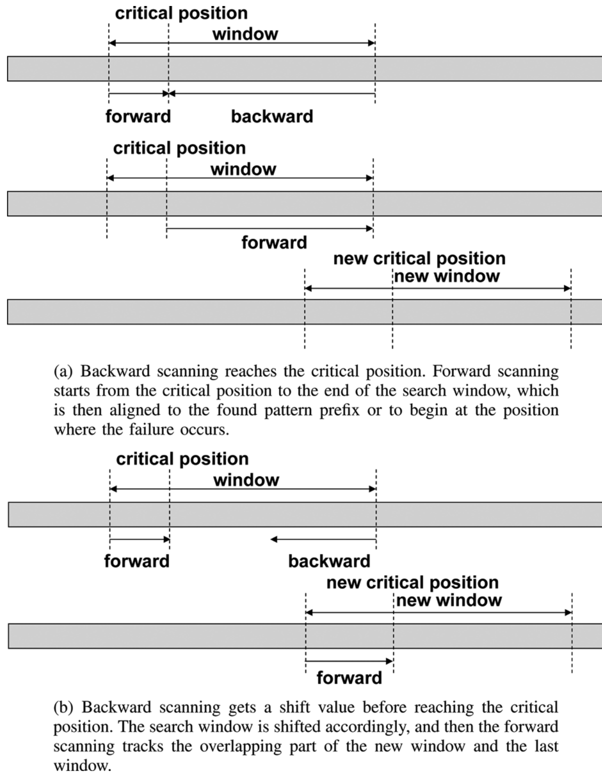


Fig. 3. Procedure to maintain the linear time performance.

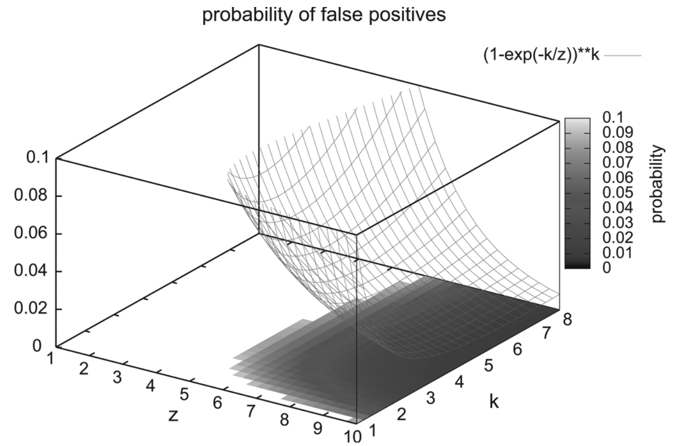


Fig. 4. False-positive rate with respect to k and the ratio of $z = v/r$.

Although the second alternative can guarantee linear worst-case time and offer sub-linear time performance on average, it has two overheads. First, it needs the space to store the AC automaton for forward scanning. Second, it needs forwarding scanning in each shift to exclude blocks from being revisited by backward scanning, but the forward scanning is an overhead if the backward scanning gets a shift value before reaching the critical position. This alternative will slow down the average performance due to the overheads. We thus suggest it be used when the linear time guarantee is a must. In this paper, we leave the second alternative optional, and implement only the BFAST architecture and the verification module to be introduced in Section IV.

E. Hash Functions and the Parameters in the Design

The hash functions in the Bloom filters are from a slight modification to a class of universal hash functions, namely the H_3 class of functions. Let $\mathsf{X} = \{0, 1, \ldots, 2^c - 1\}$ be a set of key values in c bits and $\mathsf{V} = \{0, 1, \ldots, 2^\nu - 1\}$ be addresses of a bit vector of ν bits. It is presented in [34] that the H_3 class has uniform mapping, meaning that the probability of a hash function mapping a key to a specific position is $1/2^\nu$. The implementation is also very simple.

Consider the block X as a bit string of $\langle x_1, x_2, \ldots, x_c \rangle \in \mathsf{X}$. This work defines the hash function $h_d : \mathsf{X} \longrightarrow \mathsf{V}$ for the Bloom filters by

$$h_d(X) = d_1 \bullet x_1 \oplus d_2 \bullet x_2 \oplus \cdots \oplus d_c \bullet x_c \oplus d_{c+1} \quad (5)$$

where $\bullet$ is an AND operator, $\oplus$ is a bitwise XOR operator, and d_i is a random number ranging from 0 to $2^\nu - 1$. Each of the k hash functions in a Bloom filter chooses a different set of d_i. We add an extra term d_{c+1} in this equation because if a block X contains all zeros, the k hash functions will all map the block to zero. Hence the false-positive rate will depend only on bit 0 of the bit vector, and the benefit of using k hash functions will be voided.

The modification keeps the uniform mapping of the H_3 class. Let X' be a set of strings of $c+1$ bits. The hash function $h'_d : \mathsf{X}' \longrightarrow \mathsf{V}$, where

$$h'_d(X') = d_1 \bullet x'_1 \oplus d_2 \bullet x'_2 \oplus \cdots \oplus d_c \bullet x'_c \oplus d_{c+1} \bullet x'_{c+1} \quad (6)$$

is a function in the H_3 class by definition, so the mapping to V is uniform. Since the key space of h_d, i.e., X, can be viewed as a subset of X', where x'_{c+1} is always 1, the mapping of h_d is also uniform.

Properly choosing k and the ratio of v/r can control the false-positive rate of a Bloom filter, $f = (1 - e^{-rk/v})^k$. Although setting $k = (v/r)ln2$ can minimize the probability to $f = (1/2)^k$ [10], the hardware complexity and simultaneous memory accesses also increase with a large k. Fig. 4 presents the false-positive rate with respect to k and v/r. Considering a good balance between the hardware complexity and the false-positive rate, we arbitrarily choose $k = 4$ and v/r to be around 10 so that f is around 0.012, which is low enough in practice. If the memory space is restricted, v/r can be reduced at the cost of higher false-positive rate.

If the block consists of only one or two characters, the probability that it is a pattern suffix is high, let alone the chances of occurrence in other positions of the patterns. Therefore, the Bloom filters are likely to report a hit, and the shift distance is generally short. A larger block size can reduce the probability, but it also complicates matching a pattern shorter than b, as every substring of the block should be matched against the patterns in G_0 [See the discussion below (3)]. We arbitrarily choose $b = 4$ herein because it fits well on a 32-bit data bus and is a good balance. The probability is tiny for a block to appear in a specific position of the patterns (or equivalently, a group), say around 7×10^{-6} for $r = 30\,000$.

F. Analysis of the BFAST Algorithm

We consider the performance when the characters are uniformly distributed in the analysis. The probability that a block

is in a group $G_j,\ j = 0\cdots m-b+1$ is tiny, so the probability of a hit in a Bloom filter is approximately the false-positive rate f. To simplify the analysis, we assume that an additional check for B_{j+1} is performed only when $BF(G_j)$ reports a hit for B_j. The assumption will underestimate the shift value because the *bad-block* heuristic is more aggressive, but it is sufficient to show the time complexity of the BFAST algorithm.

Let S_m denote the expected shift value for the shortest pattern length m, and $\mathcal{P}(s=i)$ denote the probability that the shift value s by querying from a single block is i. S_m can be recursively derived by

$$S_m = \begin{cases} \mathcal{P}(s=0)S_{m-1} + \sum_{i=1}^{m} \mathcal{P}(s=i)i, & \text{for } m > b \\ \sum_{i=1}^{m} \mathcal{P}(s=i)i, & \text{for } m = b. \end{cases} \quad (7)$$

If $s=0$, additional checks will determine the shift value; otherwise, the search window is shifted by s. We consider only the case that $m \geq b$ for simplicity. The case that $m < b$ can be derived similarly. $\mathcal{P}(s=i)$ is derived considering the following three conditions.

1) B_0 is a factor of some pattern in P'. The shift value is derived according to the position of the right-most occurrence of B_0, so

$$\mathcal{P}(s=i) = (1-f)^i f, \quad \text{for } i = 0\cdots m-b. \quad (8)$$

2) B_0 is not a factor of any pattern in P', but a suffix of B_0 is the prefix of some pattern. Let the longest length of such a suffix be $m-i,\ i = m-b+1\cdots m-1$

$$\mathcal{P}(s=i) = (1-f)^{m-b+1}(1-\mathcal{P}(N_i)) \times \prod_{j=m-b+1}^{i-1} \mathcal{P}(N_j), \ \text{for } i = m-b+1\cdots m-1 \quad (9)$$

where $\mathcal{P}(N_j)$ is the probability that $BF(G_j)$ reports no hit from B_0's suffix of $m-j$ characters, and

$$\mathcal{P}(N_j) = 1 - \left(f + \left(1 - \left(1 - \frac{1}{|\Sigma|^{m-j}}\right)^r\right)\right). \quad (10)$$

3) Neither B_0 is a factor of any pattern in P' nor its suffix is a prefix of any pattern. In this case

$$\mathcal{P}(s=m) = 1 - \sum_{i=0}^{m-1} \mathcal{P}(s=i). \quad (11)$$

If the false-positive rate f is low enough, the probability in (8) will be small, meaning most shifts can be at least $m-b+1$. For long patterns such that $m \gg b$, the shift values are close to m, so the sub-linear time of $O(n/m)$ can be expected for random text and patterns. As mentioned earlier, the worst-case performance is $O(mn)$, but it is unusual in practice. One of the two options we propose can ensure the linear time complexity in the worst case.

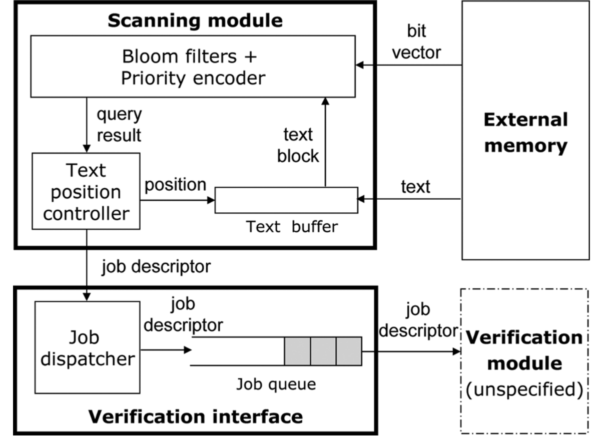


Fig. 5. Overview of the modules in the BFAST architecture.

IV. Implementation Details

A. Main Components in the BFAST Architecture

Fig. 5 presents the following three main components in the BFAST architecture.

1) *Scanning module* reads the text into the buffer, selects the block to query the Bloom filters, shifts the search window according to the querying results, and requests for verification of a suspicious match.
2) *Verification interface* receives a verification job packed in a descriptor and fills an entry in the job queue.
3) *Verification module* reads a job from the queue and performs the verification.

The sub-modules in the scanning module are described as follows.

- ***Text buffer***. The text buffer on the embedded memory loads the text from the external memory in batch. Two buffers are in the system to hide the latency in text transfer. While one buffer is being scanned, the other is loaded with the next batch of text. Because a pattern in P' may span two contiguous batches, the last $m-1$ characters in the last batch are prepended to the current batch in the buffer. Therefore, a pattern will not be missed if this case occurs.
- ***Bloom filters plus priority encoder.*** Parallel queries to the Bloom filters mean simultaneous access to the memory. In our prototype system of Xilinx XC2VP30 are 136 dual-port 18 kb memory blocks that can be accessed independently [35]. Fig. 6 illustrates the layout of the memory blocks to support multiple Bloom filters. Each memory block is configured as a 16 k×1 bit vector. The dual-port architecture can support simultaneously accessing two hash values, so two sets of memory blocks to simultaneously access $k=4$ hash values in a Bloom filter. The priority encoder determines the shift value according to the reports from the Bloom filters and evaluates whether more checks should be done. Fig. 6 skips the detail of the logics for inferring the shift distance for simplicity.
- ***Text position controller***. The text position controller keeps the position of the search window and the block for the

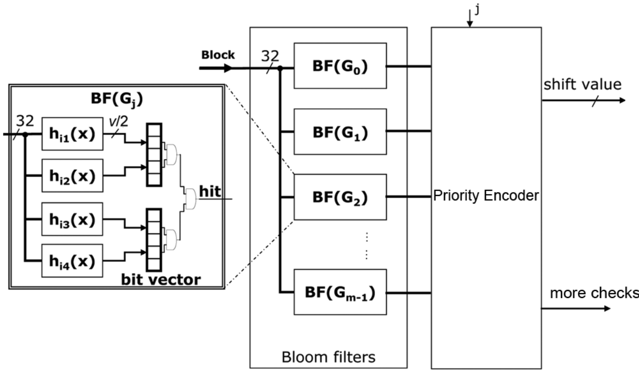


Fig. 6. Layout of memory block to support multiple Bloom filters and the priority encoder.

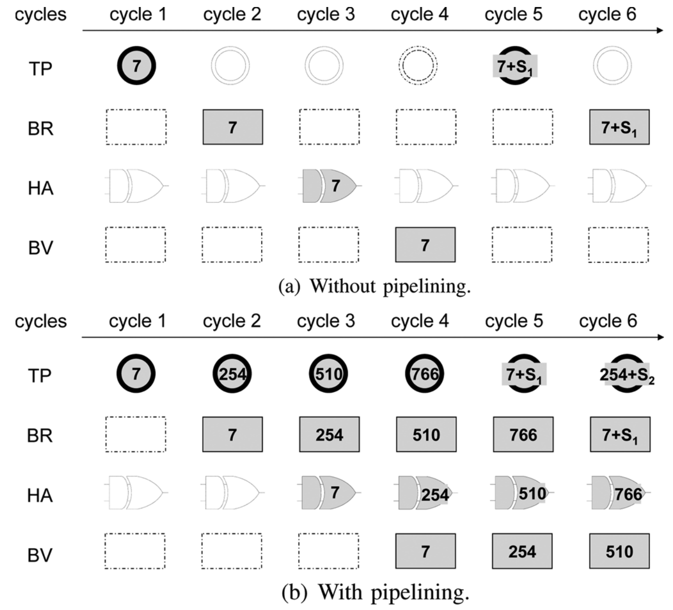


Fig. 7. Comparison of the operation without and with pipelining for $\ell = 1024$ and $m = 10$. The four phases are text position controlling (TP), block reading (BR), computing hash functions (HA), and bit vector reading (BV). S_1 and S_2 denote the shift values of the first two segments.

queries according to the feedback from the Bloom filters and the current matching status.

A non-blocking interface is located between the scanning module and the verification module. The scanning module can offload verification and move on the scanning without blocking when finding a suspicious match. This approach parallelizes the scanning and the verification to better utilize the hardware components. The detail of the verification module is left unspecified as long as it can the match in real time.

Our implementation in the verification module is an Anchored-AC algorithm that groups the patterns having the same prefix of length m into an individual trie. The prefixes serve as the keys to store these tries in a hash table. When the scanning module identifies a suspicious match, it instructs the *job dispatcher* to enter a *verification job descriptor*, including the starting position of the search window in the text, i.e., the *anchor*, and the window text into the *job queue*. If the verification module is available for the non-empty queue, it fetches a job to verify a match. The module then traverses the trie(s) indexed by the window to identify a true match. If the option to guarantee the linear worst-case time is implemented, the verification module should be modified to work with the scanning module as described in Section III-D.

B. Pipelining Design

The process of deriving the shift value and moving the search window is divided into four phases for pipelining: text position controlling (TP), block reading (BR), computing hash functions (HA), and bit vector reading (BV). The text buffer is also logically divided into four segments. Assume the buffer length is ℓ, where ℓ is a multiple of 4. The four segments are located in the ranges of $[1, \ell/4]$, $[\ell/4 - m + 2, \ell/2]$, $[\ell/2 - m + 2, 3\ell/4]$, and $[3\ell/4 - m + 2, \ell]$. Every pair of two contiguous segments overlap slightly to avoid missing a pattern in the boundary of the two segments. The overlapping part of $m - 1$ characters can ensure the patterns in P' must completely fall inside a segment. Fig. 7 illustrates the pipelining operation for $\ell = 1024$ and $m = 10$. The text position controller initializes the starting positions to the $(m - b + 1)$th character of the four segments in the beginning. The next position of the search window or the next block to be queried in the first segment is derived in the fifth cycle, that in the second segment in the sixth cycle, and so on.

V. Experimental Results and Comparisons

This work conducts a behavior simulation in C to estimate the performance, and runs a timing simulation in HDL to estimate the clock rate.

A. Simulation in C

The C simulation of the BFAST architecture was performed in four cases: 1) random patterns and text; 2) random patterns and text in Windows executable files; 3) patterns in ClamAV and random text; and 4) patterns in ClamAV and text in Windows executable files. The Windows executable files come from typical Windows applications without viruses. Because most practical files are clean and a virus signature is much shorter than the total file sizes even if it is present, using uninfected files is sufficient to estimate the performance. The ratio of v/r is set to 8 because it is close to the good compromise of 10 discussed in Section III-E. Both v and r are 2 to the power of some integer.

1) Performance for Various Number of Patterns: Fig. 8 presents the average shift values for various numbers of patterns. The values in the first three cases are all above 8, and decrease only slightly for a large pattern set because the blocks in the text are unlikely to appear in the patterns in the random cases. The values in the fourth case are degraded significantly for more than 10 000 patterns. A deep observation reveals that the blocks near or in the suffix of the patterns in P' happen to include the block “$0 \times 000 \times 000 \times 000 \times 00$”, which is frequent in some executable files. The shift values derived without the bad-block heuristic is also compared. In this case, the values start to drop significantly for more than 1000 patterns because

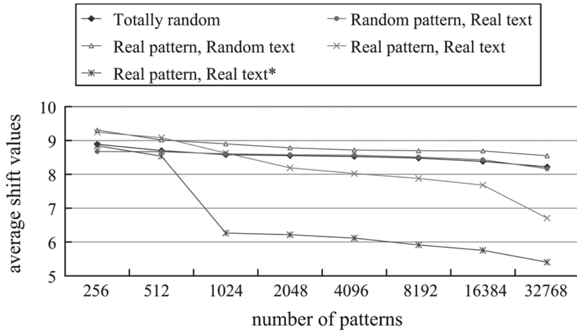


Fig. 8. Average shift values for various number of patterns in four cases. An asterisk after the "Real text" denotes the shift values are derived without the *bad-block* heuristic.

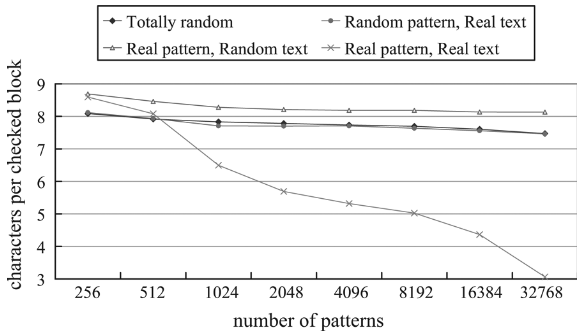
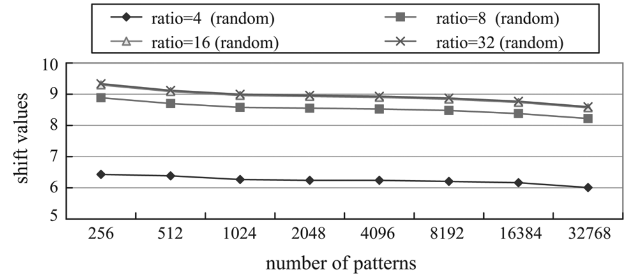


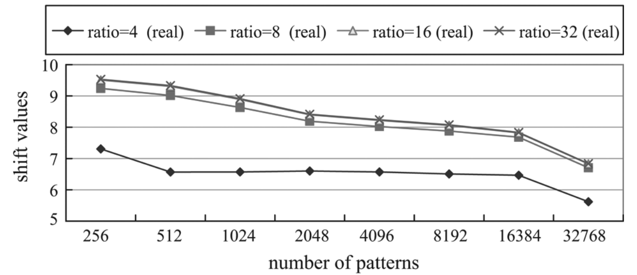
Fig. 9. Average number of characters per checked block for various number of patterns in four cases.

TABLE II
AVERAGE NUMBER OF SCANNED CHARACTERS TO MEET A VERIFICATION FOR VARIOUS NUMBERS OF PATTERNS IN THE FOURTH CASE

r	256	512	1,024	2,048
$char.$	N/A	N/A	1,156	1,111
r	4,096	8,192	16,384	32,768
$char.$	578	561	187	55



(a) random patterns and text.



(b) real patterns and text.

Fig. 10. Average shift values for various lengths of bit vectors in the Bloom filters, where $v/r = 4$, 8, 16, and 32.

the search window can advance only one character after a verification failure without the heuristic.

Deciding a shift value may need to check more than one block. Let s be the average shift value and n_s be the average number of checked blocks to derive s. The performance is estimated by $n/(sn_s)$, which is the average number of characters per checked block. Fig. 9 examines the values in the above cases. In the first three cases, the values are close to those in Fig. 8, meaning checking only one block can derive most shift values. In the fourth case, the values are degraded with increasing number of patterns. For example, 1.57 blocks are checked on average to derive a shift value for $r = 8192$, so effectively 5.03 characters are inspected in parallel.

Another concern is the verification frequency. In the first three cases, we did not find even a verification in our experiment. After all, the probability that all the blocks in the search window are in their corresponding groups (e.g., the right-most block in G_0) is tiny for $b = 4$. In the fourth case, Table II lists the average number of characters that have been scanned to meet a verification for various numbers of patterns. When $r = 8192$, for example, the verification module has a margin of around 110 cycles (given 5.03 characters inspected in parallel effectively) to verify a match without blocking.

2) Impact of the Length of the Bloom Filters: Fig. 10 compares the shift values for various lengths of bit vectors in the Bloom filters. Because the first three cases perform quite similarly, we consider only random patterns and text. The ratio of $v/r = 8$ is the best compromise between the efficiency and the memory space. This result coincides with the theoretical estimation in Section III-E. Raising the ratio up to 16 and 32 helps little to the performance, but increases the required memory space. Reducing the ratio to 4 leads to noticeable degradation. This observation justifies the choice of $v/r = 8$.

Except G_{m-2} and G_{m-1} that are stored for direct indexing and demand 65 536 and 256 bits, respectively, each of the other groups contain r substrings of the patterns. Let $z = v/r$. The total memory space required is

$$zr(m-2) + 65\,536 + 256. \tag{12}$$

For example, if $z = 8$ and $r = 10\,000$, the memory space in the Bloom filters are only around 86 kB, which can be easily accommodated in the embedded memory on a typical FPGA. The tries in the Anchored-AC algorithm are compressed in the fashion similar to that in [17]. They take 0.94 MB for 10 000 patterns, and had better be stored in the external memory.

B. HDL Simulation Result

The Xilinx XCVP30 FPGA on which the architecture is implemented has 136 dual-port embedded memory blocks, each of which can be configured as a 16 384-bit long bit vector. A set of two memory blocks work together to support 4 hash functions in a Bloom filter. Given $v/r = 8$, each set of memory blocks can store $16\,384 * 2/8 = 4\,096$ pattern blocks in a group. If $m = 10$, 8 Bloom filters store the groups from G_0 to G_7, and two more bit vectors of 65 536 bits and 256 bits store G_8 and G_9. In other words, a set of 4096 patterns takes

TABLE III
COMPARISONS BETWEEN THE BFAST AND OTHER ARCHITECTURES

category	Filtering					Finite Automata	
architecture	BFAST	[10], [39]	[40]	[8]	[11], [12], [41]	[24]	[7]
target application	anti-virus	IDS					
# of Bloom filters[a]	m	$G(t-m+1)$		k^2	N/A	N/A	N/A
# of text characters read in parallel	$b(=4)$	$G(t-m+1)$		k^2	variable	1	8
max. shift value	m	G		k	2[e]	1[e]	8[e]
# of patterns in the implementation[b]	12,288	35,475	≈2,000	2,259	≈1,500	1,000	180
embedded memory space	832 kb	400*4 kb[c]	40 kB	754 kb	288∼612 kb	0.4 MB	≈1000 kb
external memory space	≈1 MB	unavailable	144 KB	a few MBs	0	0	0
platform	Xilinx Virtex II XC2VP30	Xilinx Virtex 2000E	unavailable	Xilinx Virtex 4	Xilinx Virtex 2/4 , Spartan 3	unavailable	Xilinx XC2V6000
logic cells	7,560	16,056	unavailable	unavailable	5,219∼12,106[f]	unavailable	unavailable
number of parallel engines	1	4	1	8	2	# of split automata	1
operating frequency (MHz)	150	62.8	unavailable	250	≈ 330	unavailable	≈ 100
throughput (Gbps)	5.64[d]	2	unavailable	14.1	1.7∼5.7	10	≈ 8

[a] Notations— G: number of parallel engines, t: maximum pattern length (may be a threshold), m: minimum pattern length, k: number of characters inspected in parallel.
[b] The values can be higher if more memory space is allocated.
[c] Each parallel engine takes 400 kb.
[d] The value can be higher with longer m. The worst-case throughput is 1.2 Gbps if the signatures are properly specified, as discussed in Section III-D.
[e] The value can be larger with higher hardware cost.
[f] The values are for the implementations that filter two characters at once.

$2*8=16$ memory blocks for the groups from G_0 to G_7, and $65\,536/16\,384+\lceil 256/16\,384\rceil=5$ memory blocks for G_8 and G_9.

It is suggested that a BFAST scanning module handles around 10 000 patterns to keep high efficiency, so we allocate $3*16+5=53$ memory blocks to store $3*4096=12\,288$ patterns since 12 288 is close to 10 000. We allocate 64 memory blocks to the two text buffers, each of which takes 64 kB. The data structure in the verification module is stored in the external memory to avoid the restriction in memory space. A larger pattern set can be split into several subsets of 12 288 patterns, and multiple BFAST scanning modules, each responsible for a subset, can scan in parallel for the match. The strategy is feasible given a large FPGA, say Xilinx XC2VP100, which has 444 embedded memory blocks [35].

The system can operate at 150 MHz. The design utilizes 7560 logic cells, which amount to 24% of the available logic cells on the XC2VP30 FPGA. Given an average of 4.7 characters inspected effectively in parallel for 12 288 patterns (see Fig. 9), the throughput of the scanning module is up to $150*4.7*8=5.64$ Gb/s. If the signatures are properly specified, as discussed in Section III-D, the worst-case throughput is $150*1*8=1.2$ Gb/s. It is suggested that long signatures of at least 15 characters be used in virus-scanning applications to avoid false positives [36]. In that case, the throughput could be higher because more characters are inspected per checked block for the long signatures.

C. Comparisons With Other Works

We categorize existing designs into filtering-based and automata-based architectures. Table III summarizes their characteristics. Due to the limited table space, we leave the comparisons with some designs only in the text, but do not list them all in the table.

1) Compared With Filtering-Based Architectures: A filtering-based architecture maps a search window in the text using hash functions (including Bloom filters) to exclude the characters not in a match and verify only suspicious matches. In [10] and [37], inspecting G positions in parallel needs G sets of $L_{\max}-L_{\min}$ Bloom filters, where the pattern lengths range from $L_{\min}$ to $L_{\max}$. Because the lengths may range from a few characters to hundreds, implementing so many Bloom filters is impractical. Splitting a long string into substrings of t characters and seeking the partial matches can solve the problem [38], but the number of Bloom filters is still $G(t-m+1)$ (e.g., 96 Bloom filters for $G=4$ in [38]). G is constrained by reading so many strings from the text simultaneously for parallel queries. The throughput is only around 2 Gb/s with four parallel engines. The design also requires more logic cells than the BFAST architecture.

Dharmapurikar and Lockwood combine Bloom filters with an NFA representing the patterns[2] [8], and use the pairs of (q,x) as keys to index a hash table for the next states and failure links, where q denotes the current state and x denotes a string of at most k characters. Tracking the NFA by k characters at once involves looking for the longest match of the next k characters in the table. The design assumes real matches are rare and Bloom filters can exclude unsuccessful searches. Because a match may fall across a k-character boundary, k state machines are deployed, each of which starts at the offset of one more character from the beginning of the text. Generally, inspecting k characters in parallel requires k^2 Bloom filters, and k^2 characters are read from the text buffer simultaneously because each state machine reads k characters at once. The number of Bloom filters and the characters read in parallel will grow fast as k increases, while in the BFAST architecture, the number of Bloom filters is *linear* to the number of characters effectively inspected in parallel and only $b=4$ characters are read in each iteration. The design uses similar amount of memory to ours, even though only around 2000 patterns are inside.

[2]The method is a combination of finite automata and filtering. We arbitrarily discuss it in the filtering-based category.

Papadopoulos and Pnevmatikatos [11] use cyclic redundancy check (CRC) functions generated from the patterns as the hash functions. A replicated structure is in charge of each pattern length. Splitting long patterns into several short ones and reusing structures for the short patterns can reduce the number of structures. This strategy may be rather complex for a pattern set of many long patterns, such as that in ClamAV. Sourdis *et al.* [12] select a unique substring from each pattern, and extract only necessary bits from the substrings that can distinguish themselves from others. The bits in the text are mapped with a perfect hash function for information of a pattern. The perfect hash functions exist by grouping the patterns so that each pattern has unique bits to distinguish itself from the others in the same group. This design avoids the problem with long patterns, but the hash trees should be replicated as many copies as the number of groups. Both designs can filter two characters at once, but the replicated structures increase with the number of characters. The required logic cells for filtering only two characters at once are more than or slightly fewer than those of the BFAST architecture, let alone more characters at once. An ASIC implementation also could be a problem, as the hash functions cannot be reconfigured with updated patterns.

The Trie Bitmap Content Analyzer (TriBiCa) that builds a trie structure to identify whether a window of characters belong to a set or not [28]. TriBiCa features member identification that can indicate the matched member if the window is in the set, so the verification becomes trivial when a suspicious match is found. Because the window advances only one character at once, a single matching engine achieves the throughput of 2.5 Gb/s at 300 MHz. Four engines should work in parallel can achieve 10-Gb/s throughput. The Snort signature set in this design takes around 540 kb of memory.

2) Compared With Automata-Based Architectures: We compare the BFAST architecture with the automata-based architectures in terms of the solutions to inspecting multiple characters at once. Tan and Sherwood [22] split an automaton into several small ones in the bit level. Because the number of transitions is greatly reduced with one or a few input *bits*, expanding the automata to track multiple characters at once is facilitated. For example, at most 16 transitions are from a state in an one-bit automaton when four input characters are read. This design is not scalable to a large pattern set. Due to the length of state encoding and the partial match vector, the patterns must be partitioned into rule modules, each of which needs circuit overhead such as decoders and multiplexers. Even though we use their suggested values to minimize the memory space, i.e., 16 patterns and 4 state machines per rule module, and 8 bits in state encoding, accommodating 12 288 patterns needs totally 768 rule modules, meaning that the input characters will be simultaneously fed to so many modules. The total memory requirement becomes 4.6 MB in their calculation method. When k characters are tracked at once, the number of next state pointers in a rule module will be exponential to k. The rule modules should be also duplicated k copies for each character offset in the block of k characters. The overall cost is therefore prohibitively high for large k.

Sugawara *et al.* [7] proposed a compact data structure of the transition table for tracking multiple characters at once with hardware assistance. Their observation is that only a subset of k-character blocks and their suffixes suffice to determine the next states after k characters. However, the number of different blocks and their suffixes still increases significantly for large k in a large pattern set, making the scalability a problem. They tested the design on three rather small pattern sets of at most 180 patterns, and the table size is nearly 600 kb for only 180 patterns with $k = 4$, let alone a much larger pattern set.

Tseng *et al.* [24]–[27] build root-index tables to derive the next state after several characters from the root state, and prehashing tables to find a failure in the other states that leads to the root state. However, it has two limitations in scalability.

1) Concatenating the addresses in the root-index tables may lead to a long address to index the next table. For example, if each index table takes 8 bits to encode the characters in a given position, tracking four characters needs a 32-bit address (from four index tables), meaning 4G entries in the next table.
2) A failure in the tracking is unlikely to go back to the root state for a large pattern set, since the input character leading to a failure is likely to be the first character of some pattern, and the state transition should go to the next state from the root state according to the input character.

Therefore, the benefit of tracking multiple characters from the root state diminishes.

The work of Lunteren [23] features high efficiency in storage by compressing the AC automaton in a B-FSM data structure, which contains transition rules for fast lookup, given the current state and input character. Running from 100 to 125 MHz, a single B-FSM that reads only one input character for each state transition can achieve from 0.8 to 1 Gb/s. The design relies on aggregating the processing rate from multiple data structures of transition rules.

3) Linear Time vs. Sub-Linear Time: We do not intend to compare with all of existing architectures over the years, but discuss the pros and cons of realizing a sub-linear time algorithm in hardware. The longer the patterns, the longer the distance that the search window can slide in a sub-linear time algorithm, while the cost of a long shift distance is low. The antivirus applications typically have long patterns [36], so they could be a good target application. In applications such as Snort, the patterns may be as short as only one character. The BFAST architecture can still work for short patterns, but its performance is not optimal. This is a general weakness of a sub-linear time algorithm, which cannot skip longer than the shortest pattern length without inspection, or it may miss a match. However, a very short pattern has its own pitfall. An obvious problem is that false positives are likely to occur. Although verifying the context information in the rules can reduce the number of false positives, an attacker can infuse the short patterns in the text to force frequent verification. Therefore, we believe such performance degradation is common for existing designs, and should be addressed in research beyond string matching.

Another often criticized weakness is the worst-case performance. An attacker can exploit the weakness with an algorithmic attack. Although it is possible to implement a design that normally performs in sub-linear time and keeps in linear time in the worst case, as we have demonstrated

in Section III-D, the design has additional overheads. Reducing the overheads deserves future study. In comparison, a linear-time algorithm can guarantee the worst performance, but at the cost of replicated hardware components for parallel matching and thus higher limitation in scalability. There is a tradeoff between scalability and the need of deterministic performance. For most ordinary traffic, BFAST has an edge over other architectures. Unlike most of existing designs aiming at 2000 $\sim$ 3000 patterns in Snort, BFAST can support more than 10 000 patterns in a single engine. In summary, we believe such sub-linear time algorithms in hardware is promising, just like those in some software packages.

VI. Conclusion and Future Work

This work designs the BFAST architecture using Bloom filters to realize a sub-linear time algorithm in hardware. It can inspect multiple characters at once in effect based on algorithmic heuristics to boost the throughput up to 5.64 Gb/s for more than 10 000 virus signatures, while the worst throughput is 1.2 Gb/s with properly specified signatures. The architecture needs only m Bloom filters and reads a block of only $b = 4$ characters from the text per iteration, and features low hardware cost and memory usage for high throughput. Although a method to guarantee linear worst-case time complexity is proposed, a more lightweight solution to reduce the overheads deserves further study in the future.

An increasing number of signatures are represented in regular expressions. This architecture can support regular expressions by filtering the text with necessary substrings in the regular expressions. The presence of a regular expression is verified only if the substrings in it are all found. This *filtration-then-verification* method is common in open-source packages such as Snort and ClamAV. Supporting regular-expression matching all in hardware is the next work we will pursue.

References

[1] G. Navarro and M. Raffinot, *Flexible Pattern Matching in Strings*. Cambridge, U.K.: Cambridge Univ. Press, 2002.

[2] Y. H. Cho, S. Navab, and W. H. Mangione-Smith, “Specialized hardware for deep network packet filtering,” in *Proc. 12th Int. Conf. Field Program. Logic Appl. (FPL)*, La Grand Motte, France, Sep. 2002, pp. 452–461.

[3] I. Sourdis and D. Pnevmatikatos, “Pre-decoded CAMs for efficient and high-speed NIDS pattern matching,” in *Proc. 12th Ann. IEEE Symp. Field-Program. Custom Comput. Mach. (FCCM)*, Napa Valley, CA, Apr. 2004, pp. 258–267.

[4] M. Aldwairi, T. Conte, and P. Franzon, “Configurable string matching hardware for speeding up intrusion detection,” *ACM SIGARCH Comput. Arch. News*, vol. 33, no. 1, pp. 99–107, Mar. 2005.

[5] Z. K. Baker and V. K. Prasanna, “A computationally efficient engine for flexible intrusion detection,” *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 13, no. 11, pp. 1179–1189, Oct. 2005.

[6] A. V. Aho and M. J. Corasick, “Efficient string matching: An aid to bibliographic search,” *Commun. ACM*, vol. 18, no. 6, pp. 333–343, Jun. 1975.

[7] Y. Sugawara, M. Inaba, and K. Hiraki, “Over 10 Gbps string matching mechanism for multi-stream packet scanning systems,” in *Proc. 14th Int. Conf. Field Program. Logic Appl. (FPL)*, Antwerp, Belgium, Sep. 2004, pp. 484–493.

[8] S. Dharmapurikar and J. W. Lockwood, “Fast and scalable pattern matching for content filtering,” in *Proc. Symp. Arch. for Netw. Commun. Syst. (ANCS)*, Princeton, NJ, Oct. 2005, pp. 183–192.

[9] B. C. Brodie, R. K. Cytron, and D. E. Taylor, “A scalable architecture for high-throughput regular-expression pattern matching,” in *Proc. 33rd Int. Symp. Comput. Arch. (ISCA)*, Boston, MA, Jul. 2006, pp. 191–202.

[10] S. Dharmapurikar, P. Krishnamurthy, T. Sproull, and J. W. Lockwood, “Deep packet inspection using parallel bloom filters,” *IEEE Micro*, vol. 24, no. 1, pp. 52–61, Jan. 2004.

[11] G. Papadopoulos and D. Pnevmatikatos, “$\mathrm{Hashing} + \mathrm{memory} = \mathrm{low\ cost}$, exact pattern matching,” in *Proc. 15th Int. Conf. Field Program. Logic Appl. (FPL)*, Tampere, Finland, Aug. 2005, pp. 39–44.

[12] I. Sourdis, D. Pnevmatikatos, S. Wong, and S. Vassiliadis, “A reconfigurable perfect-hashing scheme for packet inspection,” in *Proc. 15th Int. Conf. Field Program. Logic Appl. (FPL)*, Tampere, Finland, Aug. 2005, pp. 644–647.

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

[14] B. H. Bloom, “Space/time tradeoffs in hash coding with allowable errors,” *Commun. ACM*, vol. 13, no. 7, pp. 422–426, Jul. 1970.

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

[16] N. Tuck, T. Sherwood, B. Calder, and G. Varghese, “Deterministic memory-efficient string matching algorithms for intrusion detection,” in *Proc. 23th IEEE Infocom Conf.*, Hong Kong, China, Mar. 2004, pp. 333–340.

[17] M. Norton, “Optimizing pattern matching for intrusion detection,” Sourcefire, Inc., Columbia, MD, 2004. [Online]. Available: http://www.snort.org/docs

[18] T. Lecroq, “Illustrations of the Boyer-Moore Algorithm,” Univ. Rouen, Mont-Saint-Aignan, France, 1997. [Online]. Available: http://www-igm.univ-mlv.fr/~lecroq/string/node14.html

[19] Xilinx Inc., San Jose, CA, “Two flows for partial reconfiguration: Module based and difference based,” 2004.

[20] J. Moscola, J. W. Lockwood, R. Loui, and M. Pachos, “Implementation of a content-scanning module for an internet firewall,” in *Proc. IEEE Symp. Field-Program. Custom Comput. Mach. (FCCM)*, Napa Valley, CA, Apr. 2003, pp. 31–38.

[21] I. Sourdis and D. Pnevmatikatos, “Fast, large-scale string match for a 10 Gbps FPGA-based network intrusion detection system,” in *Proc. 13th Int. Conf. Field Program. Logic Appl. (FPL)*, Lisbon, Portugal, Sep. 2003, pp. 880–889.

[22] L. Tan and T. Sherwood, “Architectures for bit-split string scanning in intrusion detection,” *IEEE Micro*, vol. 26, no. 1, pp. 110–117, Jan. 2006.

[23] J. van Lunteren, “High-performance pattern-matching for intrusion detection,” presented at the 25th IEEE Infocom Conf., Barcelona, Spain, Apr. 2006.

[24] K.-K. Tseng, Y.-C. Lai, T.-H. Lee, and Y.-D. Lin, “A fast scalable automaton matching accelerator for embedded content processors,” *ACM Trans. Embedded Comput. Syst.*, accepted for publication.

[25] Y.-D. Lin, K.-K. Tseng, T.-H. Lee, C.-C. Hung, and Y.-C. Lai, “A platform-based soc design and implementation of scalable automaton matching for deep packet inspection,” *J. Syst. Arch.*, vol. 53, no. 12, pp. 937–950, Dec. 2007.

[26] Y.-D. Lin, K.-K. Tseng, C.-C. Hung, and Y.-C. Lai, “Scalable automaton matching for high-speed deep content inspection,” presented at the 21th IEEE Adv. Inf. Netw. Appl. (AINA), Niagara Falls, Canada, May 2007.

[27] K.-K. Tseng, Y.-D. Lin, T.-H. Lee, and Y.-C. Lai, “A parallel automaton string matching with pre-hashing and root-indexing techniques for content filtering coprocessor,” presented at the 16th IEEE Int. Conf. Appl.-Specific Syst., Arch., Process. (ASAP), Samos, Greece, 2005.

[28] N. S. Artan and H. J. Chao, “Tribica: Trie bitmap content analyzer for high-speed network intrusion detection,” presented at the 26th IEEE Infocom Conf., Anchorage, AL, May 2007.

[29] S. Wu and U. Manber, “A fast algorithm for multi-pattern searching,” Dept. Comput. Sci., Univ. Arizona, Tempe, Tech. Rep. TR94-17, 1994.

[30] R.-T. Liu, N.-F. Huang, C.-H. Chen, and C.-N. Kao, “A fast pattern-match engine for network processor-based network intrusion detection system,” in *Proc. Inf. Technol.: Coding Comput. (ITCC)*, Las Vegas, NV, Apr. 2004, pp. 97–101.

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

[32] G. Navarro and M. Raffinot, "New techniques for regular expression searching," *Algorithmica*, vol. 41, no. 2, pp. 89–116, Nov. 2004.

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

[34] M. V. Ramakrishna, E. Fu, and E. Bahcekapili, "Efficient hardware hashing functions for high performance computers," *IEEE Trans. Comput.*, vol. 46, no. 12, pp. 1378–1381, Dec. 1997.

[35] Xilinx Inc., San Jose, CA, "Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete data sheet," 2005.

[36] J. O. Kaphart and W. C. Arnold, "Automatic extraction of computer virus signatures," in *Proc. 4th Virus Bulletin Int. Conf.*, Abingdon, England, Sep. 1994, pp. 178–184.

[37] M. Attig, S. Dharmapurikar, and J. Lockwood, "Implementation results of bloom filters for string matching," in *Proc. 12th Annu. IEEE Symp. Field-Program. Custom Comput. Mach. (FCCM)*, Napa Valley, CA, Apr. 2004.

[38] H. Song and J. W. Lockwood, "Multi-pattern signature matching for hardware network intrusion detection systems," presented at the 48th IEEE Globecom Conf., St. Louis, MO, Nov. 2005.

[39] Y. H. Cho and W. H. MangioneSmith, "A pattern matching coprocessor for network security," in *Proc. ACM/IEEE Des. Autom. Conf. (DAC)*, Anaheim, CA, Jun. 2005, pp. 234–239.

Po-Ching Lin (M'05) received the B.S. degree in computer and information education from National Taiwan Normal University, Taipei, Taiwan, in 1995, and the M.S. and Ph.D. degrees in computer science from National Chiao Tung University, Hsinchu, Taiwan, in 2001 and 2008, respectively.

His research interests include content networking, algorithm designing and embedded hardware software codesign.

Ying-Dar Lin (SM'06) received the B.S. degree in computer science and information engineering from National Taiwan University, Taipei, Taiwan, in 1988, and the M.S. and Ph.D. degrees in computer science from the University of California, Los Angeles (UCLA), in 1990 and 1993, respectively.

He joined the faculty of the Department of Computer and Information Science, National Chiao Tung University (NCTU), Hsinchu, Taiwan, in August 1993 and has been a Professor since 1999. He was the director of the Institute of Network Engineering during 2005–2007. He is also the founder and director of Network Benchmarking Lab (NBL), cohosted by Industrial Technology Research Institute (ITRI) and NCTU since 2002, which reviews the functionality, performance, conformance, and interoperability of networking products ranging from switch, router, WLAN, to network and content security, and VoIP. In 2002, he cofounded L7 Networks Inc., which addresses the content networking markets with the technologies of deep packet inspection. At NCTU, he currently directs, or codirects, Computer and Network Center (2007), NBL (2002), Realtek-NCTU Joint Lab (2006), and D-Link NCTU Joint Lab (2007). His research interests include design, analysis, implementation, and benchmarking of network protocols and algorithms, wire-speed switching and routing, quality of services, deep packet inspection, network processors and SoCs, and embedded hardware software codesign. From 2008, he is on the editorial board of *IEEE Communications Magazine* and *IEEE Communications Surveys and Tutorials*. He was on the program committee of ICCCN'07 and a program cochair of International Computer Symposium'07.

Yuan-Cheng Lai received the B.S. degree and the M.S. degree in computer science and information engineering from National Taiwan University, Taipei, Taiwan, in 1988 and 1990, and the Ph.D. degree from computer and information science, National Chiao Tung University, Hsinchu, Taiwan, in 1997.

He joined the faculty of National Cheng-Kung University, Tainan, Taiwan in 1998. He is an currently an Associate Professor with the Department of Information Management, National Taiwan University of Science and Technology, Taipei, Taiwan. His research interests include high-speed networking, wireless network and network performance evaluation, and Internet applications.

Yi-Jun Zheng received the B.S. and M.S. degrees in computer science from National Chiao Tung University, Hsinchu, Taiwan, in 2004 and 2006.

Her research interests include network security and content networking.

Tsern-Huei Lee (S'86–M'87–SM'98) received the B.S. degree from National Taiwan University, Taipei, Taiwan, the M.S. degree from University of California, Santa Barbara, and the Ph.D. degree from the University of Southern California, Los Angeles, in 1981, 1983, and 1987, respectively, all in electrical engineering.

Since 1987, he has been a member of the faculty of National Chiao Tung University, Hsinchu, Taiwan, where he is a Professor with the Department of Communication Engineering.

Dr. Lee was a recipient of an Outstanding Paper Award from Institute of Chinese Engineers in 1991. During the past years, he served as consultant of various companies to develop large scale QoS-enabled frame-based switches/routers, integrated access devices, and unified threat management Internet appliances. His current research interests are in communication protocols, broadband switching systems, traffic management, wireless communications, and network security.