Brief Contributions

A Hybrid Algorithm of Backward Hashing and Automaton Tracking for Virus Scanning

Po-Ching Lin, *Member*, *IEEE*,
Ying-Dar Lin, *Senior Member*, *IEEE*, and
Yuan-Cheng Lai

Abstract—Virus scanning involves computationally intensive string matching against a large number of signatures of different characteristics. Matching a variety of signatures challenges the selection of matching algorithms, as each approach has better performance than others for different signature characteristics. We propose a hybrid approach that partitions the signatures into long and short ones in the open-source ClamAV for virus scanning. An algorithm enhanced from the Wu-Manber algorithm, namely *the Backward Hashing algorithm*, is responsible for only long patterns to lengthen the average skip distance, while the Aho-Corasick algorithm scans for only short patterns to reduce the automaton sizes. The former utilizes the bad-block heuristic to exploit long shift distance and reduce the verification frequency, so it is much faster than the original WM implementation in ClamAV. The latter increases the AC performance by around 50 percent due to better cache locality. We also rank the factors to indicate their importance for the string matching performance.

Index Terms—String matching, automaton, filtering, virus scanning.

1 Introduction

SCANNING the content on network or storage devices for viruses involves computationally intensive string matching against a set of virus patterns. Although designing an efficient method for high-speed content inspection has sparked a number of innovations in research lately, most of them look to hardware approaches that offload string matching to a specialized hardware engine [1], especially for Snort-style intrusion detection (www.snort.org); however, as many antivirus applications run on a software environment (e.g., a commodity computer), deploying a hardware accelerator is costly and inflexible. Compared with intrusion detection, antivirus applications used to be relatively inconspicuous as a target to be accelerated. Therefore, we believe a scalable and fast string matching algorithm and its efficient software implementation are still desired for antivirus scanning.

Software implementation of string matching algorithms faces new challenges. Malware writers want to escape detection by antivirus programs. They frequently use obfuscation techniques, such as packing malware programs with packers (e.g., UPX, Themida, etc.) [2], to generate a number of variants of a malware program. Due to this tendency, virus signatures increase very fast and should be updated frequently. Antivirus applications, therefore, have much more patterns than Snort, which has only thousands of patterns. For example, ClamAV (www.clamav.net) has claimed a set of more than 200,000 patterns. Unfortunately, a large set of patterns demand large memory space to store them, so a compact data structure to improve cache locality is critical; otherwise, string matching will be slowed down due to the "memory wall"—memory access is slow [3]. This struggle makes designing fast string matching algorithms more complicated than ever.

A common class of string matching methods, such as the Aho-Corasick (AC) algorithm [4], tracks a finite automaton constructed from the set of patterns. The tracking reads only one character in the text per iteration, but this approach does not well leverage the capability of modern processor architectures, which can read 4 bytes or more in the operands of an instruction. Although some can track multiple characters per iteration for high performance, the parallelism from hardware assistance [5], [6], software implementation does not have the blessing from the hardware parallelism. Moreover, the data structure of the automaton contains the transitions from each state and the failure links, and should be compressed in a compact representation to reduce memory requirement [7], [8]. The existing compression methods have two limitations. First, many of them rely on hardware assistance for fast tracking, but their software implementation is sequential and much slower. Second, the number of patterns in antivirus applications is much larger than that in intrusion detection, and virus signatures are generally long (could be up to hundreds of characters) to avoid false positives [9], making compressing the patterns even challenging.

Another class of methods moves a search window through the text to check whether it contains a suspicious match or not [10], [11]. A suspicious match means that the search window is likely to be a substring of some pattern, and it is followed by a verification to see whether that pattern really appears or not. Assuming most of the text is legitimate, these methods can exclude the legitimate text very fast, and verify only those suspicious matches. The patterns can be represented in a compact data structure such as a shift table [11] or a Bloom filter [12]. There is a trade-off in choosing the window size [10]. A large window size is preferred because a long window is less likely to be matched without a true match, and thus, reduces the verification frequency. However, matching short patterns within a long window in each iteration is inefficient, since the patterns will be compared with the substrings in the long window. Some methods in the class can accelerate the scanning by skipping the characters not in a match based on algorithmic heuristics from a block of characters within the search window, such as the Wu-Manber (WM) algorithm [11]. They are generally fast, but have the Achilles' heel—the maximum skip distance (also the search window) is bounded by the shortest pattern length in the set of patterns. These methods, therefore, have the problem with short patterns, whereas short patterns do not represent a problem for automaton-based methods.

According to the above observation, either class of methods has its limitations. In other words, not a single class of methods can scan for all the virus signatures efficiently. It is, therefore, tantalizing to combine the merits of both classes to keep the data structure of the patterns manageable and fast scan through the text, while handling short patterns well. This work presents a hybrid method that combines the AC algorithm and a variant of the WM algorithm, namely the backward hashing (BH) algorithm. The patterns of virus signatures are partitioned into long and short ones, separated by a length threshold. The BH algorithm scans for only long patterns to derive long shift distance of the search

- *P.-C. Lin is with the Department of Computer Science and Information Engineering, National Chung Cheng University, 168 University Rd., Min-Hsiung., Chiayi, 621, Taiwan. E-mail: pclin@cs.ccu.edu.tw.*
- *Y.-D. Lin is with the Department of Computer Science, National Chiao Tung University, 1001, Da-Hsueh Rd., Hsinchu City, 300, Taiwan. E-mail: ydlin@cs.nctu.edu.tw.*
- *Y.-C. Lai is with the Department of Information Management, National Taiwan University of Science and Technology, 43, Sec. 4, Jilong Rd., Da-An District, Taipei City, 106, Taiwan. E-mail: laiyc@cs.ntust.edu.tw.*

Manuscript received 14 Jan. 2009; revised 22 Jan. 2010; accepted 30 Mar. 2010; published online 15 Apr. 2010.
Recommended for acceptance by P. McDaniel.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-2009-01-0023.
Digital Object Identifier no. 10.1109/TC.2010.95.

window, while the AC algorithm scans only the relatively small set of short patterns.

The backward hashing mechanism is an incremental improvement over the WM algorithm in that it can look backward from the end of the search window to verify whether a true match occurs, while exploiting long shift distance if there is a chance. A large shift is preferred, since the search window can skip more characters not in a match, and thus, speed up the search. The BH algorithm can reduce the impact on performance due to short shift distance and frequent verification due to the nonuniform character distribution in both the patterns and the text. The hybrid method, which is applied to ClamAV to improve its performance, can make the data structure of the automaton compact, and thus, save memory space. This work also examines several factors in software implementation such as cache locality, and compares their significance to the overall performance in practice.

The rest of the paper is organized as follows: Section 2 reviews the existing work for string matching for virus scanning. Section 3 presents the details of the hybrid method and the practical implementation issues, followed by the performance evaluation of the algorithm in Section 4. Finally, Section 5 concludes the paper and points out future work.

2 Review of Existing Work

This section briefly reviews the background related to this work, including existing string matching algorithms and the inner working of ClamAV on which this work is based. string matching algorithms have been developed for a diversity of applications for decades. To not distract the attention, we restrict to algorithms that have been usually used for security applications (e.g., intrusion detection and virus scanning) herein. For foundational work of string matching and other applications, we refer the readers to books such as [13] and [14]. Security applications usually involve a large number of patterns to be matched, so algorithms that scan for a single pattern at a time, e.g., the Boyer-Moore algorithm [15], are too slow for these applications. In this context, we therefore review the algorithms that scan for a large number of patterns simultaneously for speed.

2.1 String Matching Algorithms for Security Applications

Algorithms that scan the text for multiple patterns typically track the partially matched prefixes with a finite automaton that accepts the patterns, or filter the text with a search window along the text to weed out unsuccessful matches and verify only suspicious matches. The former ones read the characters in the text sequentially to drive the automaton transition, and feature linear execution time even though algorithmic attacks are present to exploit the worst case of an algorithm. But the data structure of the automaton usually demands large memory space to store the transition information. The latter ones leverage a memory-efficient hashing mechanism to fast check whether the search window contains a substring of one of the patterns. Such algorithms can execute very fast, but must carefully deal with possible algorithmic attacks.

In recent years, automaton-based approaches focus on fast tracking a compressed automaton with hardware assistance [5], [16], [17], [18], [19], but their software implementation is not as efficient as the hardware counterpart. Although some compression methods are independent of hardware [8], [20], their scalability to a large set of long virus patterns could be a problem. First, the transition table is not so sparse due to the large set of patterns. Second, the method to simplify repetitions in the patterns [20] is unable to compress the characters in the long patterns.

A filtering-based approach can map part of the search window (or the entire window) with one or more hash functions to see whether that part (or the window) matches an entire pattern or part of a long pattern. Verification for a true match follows if a suspicious match occurs. This approach is very memory efficient because a pattern is stored as only a hash value or several bits in a few addresses. The search window must be long enough to reduce the verification frequency, but has two side effects: longer time in hash computation and inability to match shorter patterns [10].

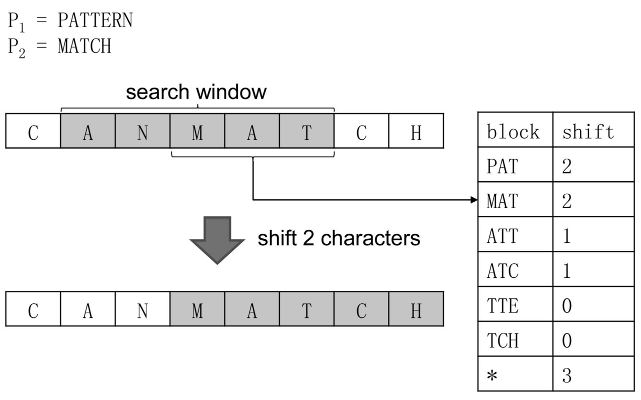

Fig. 1. A trivial example of the WM algorithm.

The WM algorithm is a well-known example of the filtering-based approach. In the search stage, the shift distance of the search window is derived by looking up the suffix of the window in the shift table (built in the preprocessing stage), and the search window can be safely shifted to the next position without missing a match. The shift is safe because of the heuristic described below.

Suppose that the search window is m characters long (set to the shortest pattern length), and let X be the block of b characters in the window suffix. The heuristic looks up the block in a shift table to derive the shift distance as follows:

1. The search window can be shifted by $m - b + 1$ characters if X is not a substring of any patterns. Any shift shorter than $m - b + 1$ cannot lead to a match because this would contradict that X does not appear in any patterns.
2. Otherwise, the shift value is $m - j$, assuming the rightmost occurrence of X ends at position j of some pattern. If $j = m$ (i.e., zero shift value), X is the suffix of one or more patterns, so whether a true match occurs should be verified. If the verification fails, the search window is shifted by one character and the process goes on.

The correctness of this heuristic was proved in [11], so we will not repeat that herein. Fig. 1 illustrates how this heuristic works with a trivial example in which the WM algorithm searches only two patterns in the text: P_1 and P_2. The shortest pattern length is five, so $m = 5$. We arbitrarily set $b = 3$ in this example. The shift table is built for substrings of $b = 3$ characters in the patterns according to the above heuristic. By looking up the suffix MAT in the shift table, the algorithm knows that the search window can be safely shifted by two characters without losing any match. In the next iteration, the shift value from the table lookup is 0, meaning that a suspicious match is found. The algorithm then verifies the occurrence of a true match.

There is a trade-off with the size of the shift table. Mapping multiple blocks to the same table entry and filling in the minimum shift value of these blocks can compress the shift table in the memory space. Reducing the table size also improves cache locality, but at the cost of smaller shift values (due to the minimum shift values) and more frequent verification (due to the increasing likelihood of a zero shift value). On the contrary, a large shift table can derive larger shift values due to fewer blocks mapped to the same entry, on average, but at the cost of worse cache locality.

With an increasing number of patterns, the hash collisions will also increase significantly, and maintaining a small table for good cache locality becomes difficult. The problem with scalability to a large set of patterns also occurs in other filtering-based algorithms [21], such as [22], [23], [24], and [28]. Zhou et al. [21] proposed a multiphase dynamic hash (MDH) algorithm that attempts to increase the scalability. This algorithm allows a compressed shift table and adds an additional hash table to separate blocks hashed to nonzero shift values from those hashed to zero shift values. This approach can reduce the chances of verification due to hash collisions that contain a zero shift value, but it is unable to reduce those due to blocks really mapped to a zero shift value. Such blocks might be frequent because of the nonuniform distribution of characters in the text and patterns in practice, and result in an amount of verification. The nonuniform distribution means that some characters or blocks appear more frequently than the average in probabilistic analysis. If a frequent block in the text happens to be the suffix of some pattern, the chances of verification will increase. This problem should be handled to further reduce the chances of verification in practice. We will look into this issue in Section 3.1. Lin et al. [28] proposed a BFAST architecture to derive good shift values for a large set of patterns using Bloom filters with hardware parallelism. However, the parallelism is not feasible in software implementation. An efficient software-based approach is still needed.

2.2 Virus Signatures and String Matching in ClamAV

The virus database in ClamAV has been growing very fast lately, containing more than 200,000 signatures at the time of writing (October 2008). The database contains four types of virus signatures: basic patterns, multipart patterns, MD5 patterns, and phishing patterns (See `clamdoc.pdf` and `signatures.pdf` in the source package). A basic pattern is simply a string of characters for exact match, while a multipart pattern consists of multiple parts of basic patterns to be matched in sequence for virus identification. The former is sufficient to detect nonpolymorphic viruses and the latter allows specifications such as wildcard characters and bounded gaps (the minimum or maximum distance between two consecutive parts) to detect polymorphic viruses. The MD5 matching computes the 16-byte (i.e., 128-bit) MD5 values from sections in the Portable Executable (PE) file and then checks whether the values match one of the MD5 patterns. The phishing matching checks whether a URL is in the URL list of phishing patterns. Matching the latter two types of patterns is much simpler than matching the others. Rather than scanning along the long file content for multiple patterns, either the MD5 value or the URL is stored in a short buffer, and the match checks *inside* the fixed buffer, simply using hashing and verification to match the MD5 patterns and tracking the automaton of phishing patterns with the characters in the buffer.

Some patterns come with contextual information such as target file type, position of the pattern in the text, and so on to reduce false positives. For example, the signature `W32.Deadc0de` is a four-character basic pattern: `0xdec0adde`. The occurrence of this pattern must start from the 64th byte in a file of PE format, or the match will not be claimed. ClamAV separates the signatures other than generic ones (i.e., not pertinent to any specific file type) into individual data structures for each target file type. Therefore, after determining the target file type of the text, ClamAV can scan the text for only the signatures associated with that type, besides the generic signatures. The current version of ClamAV (version 0.92) scans the text with both the AC algorithm and the WM algorithm.[1] The AC algorithm looks for the parts in the multipart patterns and claims a match if all the parts of a pattern appear and the relationship between the parts satisfies the specifications of the pattern. For example, the bounded gaps between two consecutive parts are all satisfied. On the other hand, the WM algorithm is responsible for matching basic patterns. Table 1 summarizes the number of basic patterns or parts (of multipart patterns). The minimum/maximum lengths of these basic patterns or parts are also listed for the two algorithms in each target type.

1. ClamAV calls it the Boyer-Moore (BM) algorithm, but the algorithm actually operates in the same way as the WM algorithm.

TABLE 1
The Number of Parts in Multipart Patterns (Scanned by AC) and Basic Patterns (Scanned by WM), as well as Their Minimum/Maximum Lengths in Each File Type

	Aho-Corasick (AC)			Wu-Manber (WM)		
File Type	num.	min	max.	num.	min.	max.
Generic	4,704	2	144	29,794	10	246
MS PE	2,878	2	176	48,852	4	392
MS OLE2	1,474	2	134	177	23	176
HTML	3,142	2	140	1,629	5	355
Mail	390	3	120	838	12	172
Graphics	2	3	26	0	N/A	N/A
ELF	0	N/A	N/A	15	17	198
Summary	subtotal	min.	max.	subtotal	min.	max.
	12,590	2	176	81,305	4	392
MD5	0	N/A	N/A	143,641	0	0
Phishing	206	6	43	0	N/A	N/A
Summary	total	min.	max.	total	min.	max.
	12,796	2	176	224,946	4	392

An old version of ClamAV simplified the AC automaton to a trie structure up to a maximum height, say, h. The patterns with identical prefixes of h characters are stored in a linked list pointed by a leaf node at level h. Because the minimum length of the patterns (actually parts in the multipart patterns) for the AC algorithm is only two characters, h was set to 2, and the linked list became increasingly longer as the set of patterns grew. Traversing the long linked list is, therefore, time-consuming. Miretskiy et al. [25] proposed a trie structure that can store a pattern at the lowest possible level as soon as the pattern's unique prefix is identified, but an automaton still needs the space to accommodate thousands of patterns. The more the patterns, the more the nodes are built in the trie. The trie structure is inherently expensive in space because each node in it must contain 256 pointers, each of which either points to the next node or is null. If a pointer takes 4 bytes, the pointers alone in a node take 1 KB space.

Erdogan and Cao [10] presented a filtering approach named Hash-AV to weed out most of the legitimate text with a Bloom filter [12], which can reside in the L2 cache due to its space efficiency. The design selects a window of seven characters with four hash functions mapped to a Bloom filter for filtering out the text not in a match. The four hash functions are applied sequentially to each window sliding along the text. If a hash function can filter out the window, the rest of the hash functions are skipped. Because the Bloom filter is unable to handle the patterns shorter than seven characters, they are left to the AC algorithm for multipart patterns. The search window in Hash-AV does not skip any characters in the text, unlike the original WM algorithm in ClamAV. Hash-AV prefers to abort the benefit of skipping because the search window is rather short in ClamAV due to the short patterns, and the short window significantly limits the skip distance. Deriving the skip distance with only three characters in ClamAV also results in high false positive rate. However, skipping a long distance is still beneficial to high performance if the search window is long. Moreover, Hash-AV does not attempt to speed up the AC algorithm in ClamAV at all.

TABLE 2
The Symbols Used in the Text

symbol	description
m	the length of the search window (also the shortest pattern length)
X	the block in the suffix of the search window to be hashed into the shift table
b	the length of X, i.e., the block size
Σ	the set of characters (We assume $\lvert\Sigma\rvert = 256$ in the text, i.e., the number of values in a byte.)

3 The Hybrid Algorithm and Practical Issues

This work partitions the patterns (either basic patterns or parts in multipart patterns) in ClamAV into long and short ones. The BH algorithm is responsible for only long patterns to lengthen the average skip distance and reduce the frequency of verification, while the AC algorithm scans for only short patterns to reduce the automaton sizes. Table 2 lists the symbols used in the text to clarify the explanation.

3.1 The BH Algorithm

The implementation of the WM algorithm in ClamAV faces several practical performance issues. First, the block in the suffix of the search window to derive the shift distance consists of only three characters (i.e., $b = 3$). Considering the huge space of $|\Sigma|^b = 256^3$ possible blocks in the window suffix, the block size seems sufficient to weed out most false positives (i.e., suspicious matches that should be verified) from a probabilistic aspect, but it is not the case in practice due to nonuniform block distribution. For example, the block "0x00 0x00 0x00" is frequent in Windows executable files. If this block happens to be the suffix of some pattern, the false positive rate due to this block will be relatively high, up to around 37 percent in our study on a sample set of Windows executable files. Extending the block size can reduce the false positive rate, but has three side effects: 1) Computing the hash function to look up a large block in the shift table takes longer time. 2) The maximum shift distance in the WM algorithm is $m - b + 1$. Increasing b will also shorten the maximum distance. 3) Although the heuristic such as that in [26] allows the maximum distance up to m characters, filling up the shift table in the preprocessing stage will be time-consuming for large b. We will discuss this point in detail later. Second, the search window in the implementation is rather short because the shortest pattern for the WM algorithm has only four characters (see Table 1). Skipping longer than the shortest pattern length may miss the shortest pattern if it happens to appear in a position between two consecutive skips. This factor restricts the effectiveness of applying the WM algorithm to ClamAV.

We use the following methods in the BH algorithm to solve the aforementioned problems.

3.1.1 A Better Heuristic to Determine the Shift Distance

The heuristic in the WM algorithm is conservative because it considers only the entire block to derive the shift distance—if the rightmost block X is not a substring of any patterns, the shift value is $m - b + 1$. The value could be larger if X's suffix is also considered. For example, if neither X appears in the patterns nor any X's suffix is a prefix of some pattern, the shift value of m is safe, i.e., no match will be missed. Liu et al. have a similar observation in their method that indexes the shift table from the prefix sliding window (PSW) [26], i.e., the prefix of the search window, but the forward search may result in false negatives. Fig. 2 illustrates an example to show that the shift should not go beyond the PSW. In this example, the shift distance derived from the above heuristic with the PSW could be $m = 5$, but the pattern `MATCH` will be missed with that shift. Generally, if the characters beyond the PSW are not examined, no possible match should be excluded.

In the hybrid algorithm, BH looks for only long patterns, so it is safe to assume $m > b$. The new heuristic is formally stated as follows:

1. If neither X is a substring of any patterns nor any suffix of X is a prefix of any patterns, the shift value can be m.
2. X is not a substring of any patterns, but it has at least one suffix that is also the prefix of some pattern. Let k be the longest length of such a suffix. The shift value can be $m - k$.
3. X is a substring of some pattern. The shift value is $m - j$, assuming that the rightmost occurrence of X ends at position j of some pattern. If $j = m$, X is the suffix of some pattern, and whether a true match occurs should be verified.

For example, if the search window in Fig. 1 contains `ANMAX`, the search window could be shifted by $m = 5$ characters without any missing match, since neither `MAX` is a substring of any patterns nor any suffix or `MAX` is a prefix of any patterns.

In the preprocessing stage, the BH algorithm builds a shift table according to the above heuristic, much similar to that in the WM algorithm. Suppose that a block X is mapped to the table with the hash function h. The steps of building a shift table are as follows:

1. Initialize each entry in the shift table to $\max(m, b)$. This value is filled because the maximum shift distance is m if $m \geq b$, and b otherwise.
2. For all $x = x_1 \ldots x_q$ that is a prefix of some pattern, where $1 \leq q < \min(m, b)$ and $x \in \sum^q$, set SHIFT$[h(yx)]$ to $\max(m, b) - q$, for all $y \in \sum^{b-q}$.
3. For every block X that is a substring of some pattern, set SHIFT$[h(X)]$ to $m - j$, where the rightmost occurrence of X ends at position j. If $b > m$, this step will be ignored because no such X exists.

If there is a hash collision, the table entry with the collision is set to the minimum of the shift values, exactly the same approach taken in the WM algorithm.

There is a trade-off with choosing the block size b. Increasing b can reduce the false positive rate, but also complicate building the shift table. Consider $q = 1$ in the second step of the above heuristic,

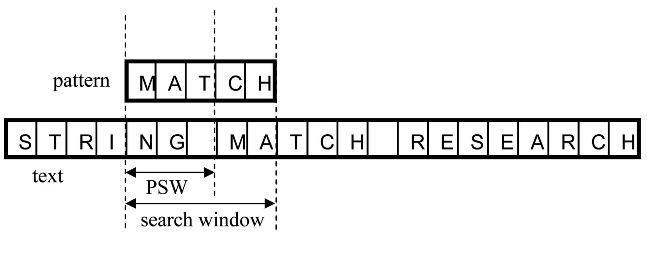

Fig. 2. The illustration of a missed match.

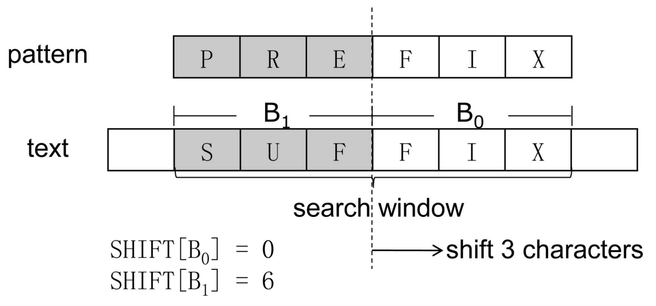

Fig. 3. The heuristic in the bad-block heuristic.

for example, we need to set $|\Sigma|^{b-1}$ entries in the shift table for each pattern. In other words, the number of entries to be set is exponential with b, making building the shift table with large b very time-consuming.

3.1.2 The Bad-Block Heuristic

The bad-block heuristic intends to reduce the false positive rate and exploit a large shift value if possible, while keeping the block size manageable. The block size is still 3, but the heuristic refrains from immediate verification of the entire search window, which is costly. The idea of this heuristic is simple. If looking up the block B_j in the shift table derives a zero shift value, where B_j denotes the jth nonoverlapping block counted backward from the suffix of the search window, it is possible to also look up B_{j+1} for exploiting a larger shift value. The entire window is verified only when looking up the blocks B_0, $B_1, \ldots, B_{\lfloor m/b \rfloor -1}$ all implies a suspicious match, i.e., SHIFT[B_j]$\leq jb$, for all $j = 0 \ldots \lfloor m/b \rfloor - 1$ (to be discussed later).

Fig. 3 illustrates the above idea with a trivial example. Suppose the algorithm scans for only a pattern PREFIX. The block B_0 in the search window matches the suffix of the pattern (derived from SHIFT[B_0]=0), so the lookup goes on to the next block B_1. In this example, SHIFT[B_1]=6, which implies that B_1 is neither a substring of the pattern nor any of B_1's suffixes is a prefix of the pattern. Note that the values in the shift table are derived based on B_0, so they are not applicable to B_1. The distance from B_1 to B_0 should be deducted from the values to meet the reasoning in the WM algorithm. Therefore, the safe shift distance is $6 - 3 = 3$, since B_1 is three characters away from B_0.

Two subtleties are in the heuristic: 1) To compress the shift table, multiple blocks are mapped to the same entry in which the minimum shift values of them are filled. The shift value derived in the heuristic may be smaller than it should be, but it is still safe—no match will be missed. 2) The shift table does not keep the exact information such as whether a block appears in a specific position or appears multiple times in the patterns. Losing the information will sometimes result in suboptimal shift values or unnecessary verification. Fig. 4 illustrates an example in which SHIFT[B_0]=0 because DNS is the suffix of the pattern. When the bad-block heuristic looks up B_1 in the shift table, it only knows that DNS is in the suffix of the pattern. Whether DNS also appears at

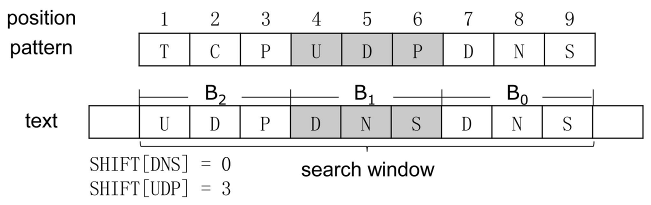

Fig. 4. An example to illustrate the information lost in the shift table.

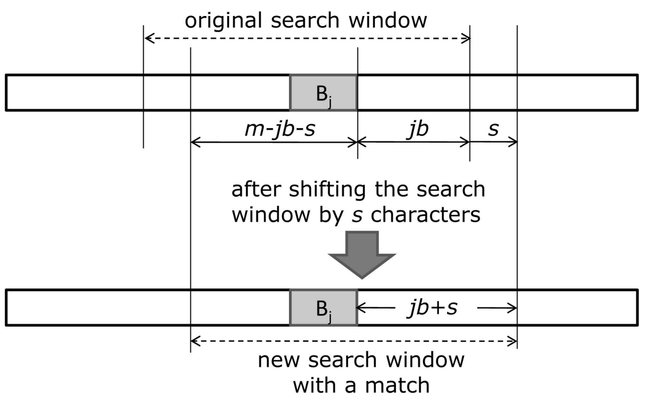
original search window
Bj
m-jb-s
jb
s
after shifting the search window by s characters
Bj
jb+s
new search window with a match

Fig. 5. The scenario in proving correctness of the bad-block heuristic.

position 4-6 is unknown from the table. The heuristic therefore has to look at B_2 further, even though a shift of 6 characters is safe in this case.

In general, if SHIFT[B_j] $> jb$, a shift of SHIFT[B_j] $- jb$ characters is safe; otherwise, the bad-block heuristic is unable to determine a shift value, and had better keep on looking up B_{j+1} for not missing any match. If a shift value larger than 0 cannot be determined from all the blocks $B_0, \ldots, B_{\lfloor m/b \rfloor -1}$, the entire search window is then verified.

The correctness of the bad-block heuristic is proved as follows:

Theorem 1. *The shift value derived in the bad-block heuristic is safe. That is, if $SHIFT[B_j] > jb$, a shift of $SHIFT[B_j] - jb$ characters is safe.*

Proof. Suppose a match occurs when the search window is shifted by s, where $s < \mathrm{SHIFT}[B_j] - jb$. This means that there is a block B_j ending at position $m - jb - s$ of the matched pattern. SHIFT[B_j] is then set to $m - (m - jb - s) = jb + s$ according to the heuristic described in Section 3.1 (also the heuristic in the WM algorithm), which says that the shift value is $m - k$, assuming the rightmost occurrence of X ends at position k of some pattern.[2] SHIFT[B_j] may be smaller due to hash collisions in table compression or another block with the same content closer to the suffix of some pattern than B_j. The reasoning implies that

$$\begin{aligned} \mathrm{SHIFT}[B_j] &\leq jb + s \\ &< jb + \mathrm{SHIFT}[B_j] - jb \\ &= \mathrm{SHIFT}[B_j]. \end{aligned} \tag{1}$$

Equation (1) leads to a contradiction, i.e., if the search window is shifted by less than SHIFT[B_j] $- jb$, no match should occur. Therefore, a shift of SHIFT[B_j] $- jb$ is safe. Fig. 5 helps to visualize the scenario in the proof. □

3.2 The Hybrid Method

The performance of the BH algorithm is subject to the shortest pattern length, so we leave out the patterns shorter than ℓ to the AC algorithm. On the contrary, patterns longer than or equal to ℓ for the AC algorithm are left to the BH algorithm. This approach is feasible because both algorithms track with basic patterns without special characters such as wildcards, which are left to the verification stage. The difference is that the AC algorithm needs to track whether the individual parts in the multipart patterns are matched in the sequence as specified.

2. We deliberately use the dummy variable k to avoid confusion with j in the proof.

TABLE 3
The Number of Parts or Basic Patterns in Each Target Type After Sorting Them Out

	$\ell = 9$		$\ell = 12$		$\ell = 15$	
Type	AC	BH	AC	BH	AC	BH
Generic	1,011	33,487	1,335	33,163	2,038	32,460
MS PE	1,398	50,332	1,626	50,104	1,859	49,871
MS OLE2	90	1,561	145	1,506	226	1,425
HTML	452	4,319	725	4,046	854	3,917
Mail	156	1,072	175	1,053	231	997
Graphics	1	1	1	1	1	1
ELF	0	15	0	15	0	15
Total	3,108	90,787	4,007	89,888	5,209	88,686
Orig. total	12,590	81,305	12,590	81,305	12,590	81,305

In the hybrid method, the short patterns left to the AC algorithm, which treats them as a special case of the multipart patterns (i.e., only one part) and verifies the match with the optional contextual information (if any) if the single-part pattern is found. The function for tracking multiple parts in the AC algorithm can be duplicated in the BH algorithm so that the BH algorithm can track the match when the individual parts of the multipart patterns are found. Both algorithms scan the same text in turn, and the tracking information of multipart patterns (which parts have been found in which position) is shared for either to resume the tracking.

Since the block size is three, the length threshold is chosen to be a multiple of three for easy implementation. Here, we set $\ell = 9, 12,$ and 15, and sort out the patterns according to the threshold. Table 3 lists the number of parts or basic patterns in each target type after the rearrangement. In the table, the number of AC patterns is only $24.7 \sim 41.4\%$ of the original number, on average, while the patterns moved to the BH algorithm contribute just $9.1 \sim 11.7\%$ more patterns to it.

4 Parameter Selection and Performance Evaluation

4.1 Parameter Selection

The hybrid method should properly choose several parameters to optimize the performance, including the size of the shift table for good cache locality and the length threshold to separate the patterns. The following sections will discuss the choices. We run the experiments on a PC with a 1.6 GHz Xeon E5310 quad-core CPU, which has L1 cache of 64 Kbytes for each core and 2×4 MB L2 cache on the chip. Table 1 has listed the characteristics of the patterns from ClamAV version 0.92.

For the experiments, we collected a set of 13 Windows executable files of around 70 MB, such as `winword.exe` (for MS Word), `skype.exe` (for Internet phone call), and `wireshark. exe` (network protocol analyzer), since most signatures in ClamAV are generic ones and those for files of Microsoft PE format. These files come without viruses, but they are sufficient if the purpose is measuring the performance of virus scanning, not the accuracy of finding out a virus. Most executable files are clean in practice, so using normal executable files is close to the real scenario. Moreover, a virus signature is generally much shorter than the entire executable file, so its significance to the scanning speed is relatively small.

The WM algorithm in ClamAV maps a three-character block with the hash function $h(x_1, x_2, x_3) = 211x_1 + 37x_2 + x_3$, where x_1, x_2, and x_3 are the three characters in the block. The search window consists of only three characters due to the shortest pattern length.[3] According to the heuristic in the WM algorithm in which the maximum shift distance is $m - b + 1$ (see Section 2.1), where $m = 3$ and $b = 3$ in the ClamAV implementation, the shift values are very short, consisting of only 0 and 1. Around 46.6 percent of the entries are 0 in the shift table for generic signatures, and around 75.9 percent are 0 for signatures associated with MS PE format. Therefore, the average shift value is only 0.28 characters, meaning that most of the scanning time is spent in verification and the benefits of skipping in the WM algorithm are sacrificed.

3. The shortest pattern length in the WM algorithm is 4, so the implementation could be a little bit more aggressive to set $m = 4$.

TABLE 4
The Shift Values for Various Table Sizes and Length Thresholds

	0.5x	1x	2x	4x	8x
$\ell = 9$	1.56	2.22	3.10	4.1	4.93
$\ell = 12$	1.78	2.47	3.65	5.11	6.37
$\ell = 15$	1.90	2.70	4.09	5.96	7.61

We tested the performance for various table sizes and the length thresholds. The experiment controls the table sizes by tuning the hash functions. The size is roughly doubled by the hash function $h(x_1, x_2, x_3) = 422x_1 + 74x_2 + x_3$, quadrupled by $h(x_1, x_2, x_3) = 844x_1 + 148x_2 + x_3$, and so on. The larger the table size, the larger the average shift values because fewer blocks are mapped to the same table entry. However, a large table also reduces cache locality. The length thresholds are set at $\ell = 9, 12,$ and 15. Table 4 presents the shift values for various cases in the experiment, where 1x denotes the original table size in ClamAV implementation, 2x denotes doubling the size, and so on.

Two observations are from Table 4. First, scanning for only the long patterns can significantly increase the average shift distance. Second, the average shift distance is still much shorter than the maximum one (i.e., ℓ) because of the compressed table. Despite the short shift distance, on average, the frequency of verification is significantly decreased—the verification follows only when checking every block in a long search window in the backward hashing is unable to derive a shift value larger than 0.

Table 5 presents the execution time of the BH algorithm in each case. The execution time includes only that in buffer scanning to make the effect of the BH algorithm obvious. The other stages, such as buffer loading, decompression, and so on, are not counted. Although the shift values increase with ℓ, as presented in Table 4, the differences in the execution time are insignificant. The insignificance of the execution time has two implications. First, checking the blocks in a search window of $\ell = 9$ characters is sufficient to reduce the verification frequency. Increasing ℓ contributes little to reduce the frequency, and in turn, the execution time. Second, the bottleneck is the verification, but the backward hashing can effectively reduce its frequency. Combining the benefits of a long search window and backward hashing, the BH algorithm is much faster than the original implementation, which takes 14.19 seconds.

Because the AC algorithm has fewer patterns, its execution is also faster than the original implementation, but the acceleration is not so significant as that in the WM algorithm. The original AC algorithm takes 15.74 seconds to scan these executable files. The execution time becomes 10.55, 10.71, and 10.78 seconds for $\ell = 9, 12,$ and 15, respectively. The results again show that $\ell = 9$ is a proper length threshold. The AC algorithm is still much slower

TABLE 5
The Execution Time (in Seconds) for Various Table Sizes and Length Thresholds

	0.5x	1x	2x	4x	8x
$\ell = 9$	0.83	0.14	0.21	0.35	0.53
$\ell = 12$	0.83	0.14	0.21	0.33	0.49
$\ell = 15$	0.83	0.14	0.20	0.32	0.45

TABLE 6
Comparing the Throughput (Megabits per Second) between the Hybrid Method and the Original Implementation in ClamAV

(Mb/s)	The original implementation	The hybrid algorithm
Only WM/BH	41.26	4,197
Hash-AV	41.26	187.2[a](664)[b]
Only AC	37.19	55.44
WM/BH+AC	19.56	54.72

[a] The size of Bloom filter is 512 KB.
[b] An estimated value interpolated from the results from [10] with the number of patterns in the experiment.

than the WM algorithm, and a bottleneck to be further improved in the future work.

4.2 Performance Evaluation

Three major factors affect the execution time in the BH scanning: 1) shift distance, 2) cache locality, and 3) verification frequency. The three factors interact with each other. For example, increasing cache locality means smaller data structure, implying more information is "compressed," and higher verification frequency. Reducing verification frequency needs a long search window, implying the chance to exploit long shift distance.

The BH algorithm, which can reduce the verification frequency, is much faster than the ClamAV implementation, even though a large table implies worse locality. For example, when the table size is eight times larger in Table 5, the BH algorithm is still much faster than the original implementation. We can infer that reducing the verification frequency is more effective than cache locality. Also note that reducing the table size too much has a negative effect because compressing the information in the table size too much increases verification frequency, even though the locality of accessing the table increases.

Looking at the rows in both Tables 4 and 5 simultaneously, it can be seen that longer shift distance does not imply better performance because the cache locality becomes worse due to a larger shift table. But if we fix the table size, we can still see the benefit of longer shift distance when the table size is large enough, e.g., the table size 8x. Therefore, reducing cache locality is more effective than increasing the shift distance. The importance of the three factor becomes

$$\text{verification frequency} > \text{cache locality} > \text{shift distance}.$$

Note that reducing verification frequency needs a long search window. Although long shift distance looks not very beneficial, it is still a bonus and can be "piggybacked" in the implementation of a long search window. There is no reason not to allow longer shift distance for a fixed table size when we have a chance from a long search window.

Table 6 compares the throughput between the original method and the hybrid method. In the original implementation, the AC algorithm and the WM algorithm scan for polymorphic and nonpolymorphic signatures, respectively. The hybrid method changes the patterns for the AC and BH algorithms as described in Section 3.2. After the revision, the BH algorithm is 109 times faster than the original WM algorithm, due to the longer shift distance and the lower verification frequency. Despite the impressive acceleration for the WM algorithm, the improvement over the AC algorithm is only 49 percent faster. ClamAV scans the same file content in two passes with the WM/BH algorithm for generic patterns and file-type-specific patterns (see Table 1), respectively, and in another two passes with the AC algorithm for the two types of patterns. The overall throughput without counting the MD5 matching is 54.72 Mbps (the total file length divided by the total scanning time in two passes of both algorithms).

If we count the MD5 matching, according to the discussion in Section 2.2, this matching checks whether a 16-byte buffer of the MD5 value matches one of the MD5 patterns with hashing. Just a few MD5 values from sections of PE files are checked in this way. The aggregate buffer size (usually less than 100 bytes) is much shorter than an ordinary PE file, so the time of MD5 matching is relatively tiny, less than 0.1 percent of the scanning time in the BH algorithm in our experiment. However, computing the MD5 value from the file content takes around 32 percent longer time than scanning in the BH algorithm—the AC algorithm is still a bottleneck. With the MD5 matching, the hybrid method improves the overall throughput from 19.44 Mbps in the original implementation to 53.81 Mbps, which is 2.77 times faster.

We also implemented Hash-AV in [10] for comparing the throughput in Table 6. Since the processor running ClamAV, the number of patterns and the files for scanning are all different in their implementation, we also include the estimated throughput interpolated from the results of [10] for reference. The interpolation assumes the same number of 83,819 generic plus PE-type patterns (see Table 3 for $\ell = 9$) as that in the hybrid method, and derives the estimation from the throughput for 30,000 and 120,000 patterns in their outputs. In our experiment, we observed that the first two hash functions with four and six characters for hashing in Hash-AV do not filter out the text well (only 43 percent of the text is filtered out). Moreover, the throughput of Hash-AV is the best when the Bloom filter size is 512 KB, which is still larger than the shift table of around 64 KB in the hybrid method. Thus, the cache locality may not be as good. These factors can account for the difference in throughput between Hash-AV and the hybrid method. Furthermore, it is worth noting that HashAV does not introduce any improvement to the AC algorithm, which represents a bottleneck.

4.3 Discussion of Worst-Case Performance

It is theoretically possible to dramatically reduce the performance of a sublinear-time algorithm such as the BH algorithm in some extreme cases. For example, if there is a pattern aaaaa and the text consists of all as, then a verification is needed for every shift of only one character in the text, and the time complexity becomes superlinear. Although worst performance in linear time for such algorithms is possible for single string matching [27], it is nontrivial to guarantee so for multiple string matching.

Things are not so bad in practice. A virus scanner can stop the scanning after one or a few viruses are found, so the aforementioned worst case will not happen. The algorithm can also detect an algorithmic attack by counting the number of blocks that have been revisited in backward hashing. If the number of revisited blocks in a piece of text is larger than a threshold (an unusual case), an algorithmic attack is likely to happen and an alarm is raised. The piece of text that causes the alarm can be marked suspicious for further examination.

5 Conclusion

This work presents a hybrid algorithm that combines the BH algorithm for long patterns and automaton tracking in the AC algorithm for short patterns to scan the large set of virus signatures in ClamAV. The former can reduce the verification frequency and exploit long shift distance by backward hashing in the search window. It also compresses the shift table for good cache locality. The latter can effectively reduce the number of patterns in an AC automaton. The hybrid algorithm can efficiently combine the benefits of the traditional AC algorithm and the WM algorithm.

As many techniques escape detection of antivirus systems, such as packing, polymorphism, and metamorphism. Handling these evasion techniques demands more processing than ever. Therefore, those overheads are likely to eventually become new time-consuming components in an antivirus system. They will deserve further study in the future.

Acknowledgments

This work was supported in part by the National Science Council in Taiwan, and in parts by grants from Cisco and Intel.

REFERENCES

[1] P.-C. Lin, Y.-D. Lin, Y.-C. Lai, and T.-H. Lee, “Using String Matching for Deep Packet Inspection,” *Computer,* vol. 41, no. 4, pp. 23-28, Apr. 2008.

[2] F. Guo, P. Ferrie, and T. cker Chiueh, “A Study of the Packer Problem and Its Solutions,” *Proc. Int’l Symp. Recent Advances in Intrusion Detection (RAID),* pp. 98-115, 2008.

[3] W.A. Wolf and S. McKee, “Hitting the Memory Wall: Implications of the Obvious,” *Computer Architecture News,* vol. 23, no. 1, pp. 20-24, Mar. 1995.

[4] A.V. Aho and M.J. Corasick, “Efficient String Matching: An Aid to Bibliographic Search,” *Comm. ACM,* vol. 18, no. 6, pp. 333-343, June 1975.

[5] S. Dharmapurikar and J.W. Lockwood, “Fast and Scalable Pattern Matching for Content Filtering,” *Proc. Symp. Architectures for Networking and Comm. Systems (ANCS),* pp. 183-192, Oct. 2005.

[6] Y. Sugawara, M. Inaba, and K. Hiraki, “Over 10 Gbps String Matching Mechanism for Multi-Stream Packet Scanning Systems,” *Proc. 14th Int’l Conf. Field Programmable Logic and Applications (FPL),* pp. 484-493, Sept. 2004.

[7] N. Tuck, T. Sherwood, B. Calder, and G. Varghese, “Deterministic Memory-Efficient String Matching Algorithms for Intrusion Detection,” *Proc. IEEE INFOCOM,* pp. 333-340, Mar. 2004.

[8] M. Norton, “Optimizing Pattern Matching for Intrusion Detection,” technical report, Sourcefire, Inc., http://www.snort.org/docs, 2004.

[9] J.O. Kaphart and W.C. Arnold, “Automatic Extraction of Computer Virus Signatures,” *Proc. Fourth Virus Bull. Int’l Conf.,* pp. 178-184, Sept. 1994.

[10] O. Erdogan and P. Cao, “Hash-av: Fast Virus Signature Scanning by Cache-Resident Filters,” *Proc. Global Comm. Conf. (Globecom),* pp. 1767-1772, Nov. 2005.

[11] S. Wu and U. Manber, “A Fast Algorithm for Multi-Pattern Searching,” Technical Report TR94-17, Dept. of Computer Science, Univ. of Arizona, 1994.

[12] B.H. Bloom, “Space/Time Tradeoffs in Hash Coding with Allowable Errors,” *Comm. ACM,* vol. 13, no. 7, pp. 422-426, July 1970.

[13] M. Crochemore and W. Rytter, *Jewels on Stringology.* World Scientific Publishing Company, 2002.

[14] G. Navarro and M. Raffinot, *Flexible Pattern Matching in Strings: Practical On-Line Search Algorithms for Texts and Biological Sequences.* Cambridge Univ. Press, 2008.

[15] R.S. Boyer and J.S. Moore, “A Fast String Matching Algorithm,” *Comm. ACM,* vol. 20, no. 10, pp. 762-772, Oct. 1977.

[16] J. van Lunteren, “High-Performance Pattern-Matching for Intrusion Detection,” *Proc. IEEE INFOCOM,* Apr. 2006.

[17] N.S. Artan and H.J. Chao, “Tribica: Trie Bitmap Content Analyzer for High-Speed Network Intrusion Detection,” *Proc. IEEE INFOCOM,* May 2007.

[18] 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.

[19] B.C. Brodie, R.K. Cytron, and D.E. Taylor, “A Scalable Architecture for High-Throughput Regular-Expression Pattern Matching,” *Proc. 33rd Int’l Symp. Computer Architecture (ISCA),* pp. 191-202, July 2006.

[20] F. Yu, Z. Chen, Y. Diao, T.V. Lakshman, and R.H. Katz, “Fast and Memory-Efficient Regular Expression Matching for Deep Packet Inspection,” *Proc. ACM/IEEE Symp. Architecture for Networking and Comm. Systems (ANCS),* pp. 93-102, Dec. 2006.

[21] Z. Zhou, Y. Xue, J. Liu, W. Zhang, and J. Li, “MDH: A High Speed Multi-Phase Dynamic Hash String Matching Algorithm,” *Proc. Ninth Int’l Conf. Information and Comm. Security (ICICS),* pp. 201-215, Dec. 2007.

[22] B. Xu, X. Zhou, and J. Li, “Recursive Shift Indexing: A Fast Multi-Pattern String Matching Algorithm,” *Proc. Fourth Int’l Conf. Applied Cryptography and Network Security (ACNS),* June 2006.

[23] J. Kytöjoki, L. Salmela, and J. Tarhio, “Tuning String Matching for Huge Pattern Sets,” *Proc. Ann. Symp. Combinatorial Pattern Matching (CPM),* pp. 211-224, June 2003.

[24] M. Fisk and G. Varghese, “Applying Fast String Matching to Intrusion Detection,” Technical Report UCSD TR CS2001-0670, Univ. of California, San Diego, 2001.

[25] Y. Miretskiy, A. Das, C.P. Wright, and E. Zadok, “Avfs: An On-access Anti-Virus File System,” *Proc. 13th USENIX Security Symp.,* pp. 73-88, Aug. 2004.

[26] R.-T. Liu, N.-F. Huang, C.-N. Kao, C.-H. Chen, and C.-C. Chou, “A Fast Pattern-Match Engine for Network Processor-Based Network Intrusion Detection System,” *Proc. Int’l Conf. Information Technology: Coding and Computing (ITCC),* pp. 97-101, Apr. 2004.

[27] Z. Galil, “On Improving the Worst Case Running Time of the Boyer-Moore String Matching Algorithm,” *Comm. ACM,* vol. 22, no. 9, pp. 505-508, Sept. 1979.

[28] P.-C. Lin, Y.-D. Lin, Y.-J. Zheng, Y.-C. Lai, and T.-H. Lee, “Realizing a Sub-linear Time String-Matching Algorithm with a Hardware Accelerator Using Bloom Filters,” *IEEE Trans. VLSI Systems,* vol. 17, no. 8, pp. 1008-1020, Aug. 2009.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.