國立交通大學

資訊工程學系

博 士 論 文

加速深層封包檢查的字串比對演算法

Accelerating String Matching Algorithms for Deep Packet Inspection

研 究 生：林柏青

指導教授：林盈達　教授

中 華 民 國 九 十 七 年 六 月

加速深層封包檢查的字串比對演算法

Accelerating String Matching Algorithms for Deep Packet Inspection

研 究 生：林柏青　　Student：Po-Ching Lin

指導教授：林盈達　　Advisor：Ying-Dar Lin

國 立 交 通 大 學

資 訊 工 程 學 系

博 士 論 文

A Dissertation Submitted to

Department of Computer Science

College of Computer Science

National Chiao Tung University

for the Degree of

Doctor of Philosophy

in

Computer Science

June 2008

Hsinchu, Taiwan

中華民國九十七年六月

誌　　謝

還記得那時要考碩士班時，只是抱著增加經驗的心態去試試看。現在竟然也到走到了博士畢業的階段。在這段歷程中，經歷了許多人事物。沒有這麼多人的協助，恐怕也很難走到這一步。

最重要的，要感謝我的指導教授林盈達教授。從他身上學到了按部就班，有條有理的去完成一件事。以及不斷的去嘗試新的事物，提升自己的歷練。在研究上疑似無路可走的階段，能在我混亂的思緒中開挖出一條可走的路。這些能力，我想此生都會受用無窮。

感謝賴源正教授、李程輝教授。在論文的修訂上，給了很多寶貴的建議。讓我可以更順利的投出論文。

感謝我的父母，在這個漫長的階段在各方面讓我無後顧之憂，可以專心的在學業上。如果眾多事情沒有他們幫忙處理，恐怕我就很難做任何研究了。走到畢業階段，順利去工作，也了卻了他們的一個心願。

感謝旻芳，幫我招了不少論文。結婚之後，投論文的運氣好像真的有好了不少，不然還得繼續撐。在美國生活的時候，也做了多道好吃的菜，讓我自己都想開始學著做。

感謝資源超級豐富的高速網路實驗室及諸位伙伴：義能、世強、國坤、慶明、煥雲、明道、一瑋、志祥、伊君、岱穎、思豪、孟甫、碩彥、銘康、福祥、振洲、俊男、其衡、宗寰、宗憲、又賢。陪伴了我渡過這一時光，並給予相當多的協助。並感謝助理耀萱在畢業最後階段大力的幫忙。

最後也感謝 UCB 的 Vern Paxson 教授、Richard Karp 教授，給了我一些不同的想法和觀點。以及在 ICSI 的好伙伴，Robin, Christian, Nick，他們投注在工作上的精神讓人感動。Diane, Maria, Jaci, Jacob, Whitney 在 ICSI 參訪期間所給予的各項協助和支援。

加速深層封包檢查的字串比對演算法

學生：林柏青　　指導教授：林盈達

國立交通大學資訊工程學系博士班

摘　　要

為數增多的網路設備會檢視封包的內容來找尋違背安全的特徵字串。加速這個稱為深層封包檢查(Deep Packet Inspection; DPI)的程序，可從兩個方面進行：演算法及封包流程。這個研究著眼在前者。我們首先檢閱了現有用來做 DPI 的字串比對演算法，來看哪些已經做了以及哪些應該要解決的。在這個檢閱當中，我們指出每種演算法的優缺點，以及一個適合各種應用之不同的樣式集合的高延展性、有效率的設計仍是一種挑戰。我們也剖析了字串比對在各種不同樣式個數、長度及字元分佈的應用下的效能，以了解影響有效率的字串比對的重要因素。剖析的結果顯示出了哪種 DPI 的應用適合哪種演算法。在考慮了前述的議題後，我們設計了一個可以利用演算法啟示法則的硬體字串比對引擎，以及給擁有龐大樣式集合的應用，如掃毒應用的一種混合的軟體方法。然而，單是字串比對不足以加速某些 DPI 的應用，例如網頁過濾器。所以我們提出了一個機率方法，稱為早期決定演算法，來加速網頁過濾的分類。

這個研究有幾點重要的觀察與貢獻。首先，所做的檢閱和剖析研究了數個主要的 DPI 應用，包含入侵偵測、掃毒和網頁過濾，而不像大部分現有的方法集中在入侵偵測上。這個研究也顯示了字串比對在入侵偵測上並非那麼嚴重，因此字串比對的發展在其他的應用也應受到關注。所做的剖析指出記憶體的存取(因此含快取的區域性)、驗證的頻率、和搜尋視窗的移動距離是影響效能的主要因素。

其次，所呈現的字串比對硬體架構，稱之為 Bloom Filter Accelerated Sub-linear Time (BFAST)，能從 Bloom filter 中獲得演算法的啟示法則，在次線性的時間內掃描內容。BFAST 當中的 bad-block 啟示法則能比過去使用查表的方式保留精確的資訊在 Bloom filter 當中，以得到較好的效能。我們也提出了處理最壞情況的一些務實的技巧，以及一個在理論上可以達到線性時間的方法。模擬顯示出使用 8 個字元的區塊，可以讓單一字串比對引擎的效能在 16,384 個樣式下達到 9.34 Gbps。此外我們也提出了一個混合的掃描病毒方法，用於無法使用硬體方法時。我們把 ClamAV 中的樣式依他們的長短做區隔，並使用衍生自 Wu-Manber (WM) 演算法的 Backward-Hashing (BH)演算法來負責長樣式，以加長平均的移動距離，並讓 Aho-Corasick 演算法只處理短樣式以縮小自動機的大小。前者使用了 bad-block 啟示法則以獲取更長的移動距離並減少驗證的頻率，所以比 ClamAV 原先用的 WM 演算法更快。後者則因較好的快取區域性而提高了 AC 演算法約 50%的效能。

除了加速字串比對外，我們也提出了一個簡單但有效的早期決定演算法，透過只檢查一部分的網頁內容來加速過濾流程。這個演算法能儘快做出要阻擋或是通過網頁內容的決定，只要有夠高的機率來判定該內容是屬於該阻擋或是該通過的種類。實驗結果顯示這個演算法平均能只看 1/4 的網頁內容就已足夠做決定，而仍能保持相當好的準確度。這個演算法可與其他網頁過濾的方法互補來高效率地過濾網頁內容。

關鍵字：字串比對、演算法、深層封包檢查

Accelerating String Matching Algorithms for Deep Packet Inspection

Student：Po-Ching Lin　　Advisors：Dr. Ying-Dar Lin

Department of Computer Science
National Chiao Tung University

ABSTRACT

An increasing number of network devices inspect the packet content for various signatures of security violation. Accelerating the process, namely *deep packet inspection* (DPI), can be in two aspects: algorithm and packet flow. This work focuses on the former. We first review existing string matching algorithms for DPI to see what work has been done and what should be addressed. In the review, we indicate the pros and cons of each algorithm, and a scalable, efficient design to meet the different characteristics of the pattern set in various applications is still a challenge. We also profile the performance of string matching in practical applications for various numbers, lengths and character distributions of the patterns to realize the *key factors* in efficient string matching. The results shed light on which approach is appropriate for each DPI application. Considering the above study, we design a hardware-based string-matching engine that can exploit *algorithmic heuristics* for acceleration, and a hybrid software method for applications with a large pattern set, say anti-virus applications. However, it is insufficient to accelerate some DPI applications, such as Web filter, with string matching alone. We thus present a probabilistic approach, namely the *early decision* algorithm to accelerate the classification in Web filtering.

This work features a number of observations and contributions. First, the review and profiling study several major DPI applications, including intrusion detection, anti-virus and Web filtering, unlike most existing work that focuses on intrusion detection. The study shows string matching is not so critical in intrusion detection, and

its development for DPI should also cover other applications more. The profiling indicates memory access (thus also *cache locality*), *verification frequency* and *shift distance* of the search window are major factors that affect the performance.

Second, the presented hardware architecture of string matching, namely *Bloom Filter Accelerated Sub-linear Time* (BFAST), can exploit algorithmic heuristics with *Bloom filters* to scan the content in *sub-linear* time. The *bad-block heuristic* in the BFAST has better performance than that from the conventional table due to the precise information kept in the Bloom filters. We also propose practical techniques to handle the *worst case*, and the theoretical method to achieve linear worst-case time. The simulation shows that the peak throughput of a single match engine can achieve up to 9.34 Gbps for 16,384 patterns with an eight-character block in the heuristics. A hybrid method for virus scanning is also presented since the hardware approach may be not applicable. We partition the patterns in ClamAV into long and short ones. An algorithm enhanced from the Wu-Manber (WM) algorithm, namely the *Backward-Hashing* algorithm (BH), 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% due to better cache locality.

Besides speeding up string matching, we present a simple, but effective early decision algorithm to accelerate the filtering process by examining only part of the Web content. The algorithm can make the filtering decision, either to block or to pass the Web content, as soon as it is confident with a high probability that the content should belong to a banned or an allowed category. The experiments show the algorithm can examine only around one-fourth of Web content on average, while the accuracy remains fairly good. This algorithm can complement other Web filtering approaches to filter the Web content with high efficiency.

Keywords: string matching, algorithm, deep packet inspection

Table of Contents

List of Figures

List of Tables

CHAPTER 1

Introduction: string matching for deep packet inspection (DPI)

1.1 Development of string matching algorithms

A classical algorithm for decades, string matching has recently proven useful for deep packet inspection (DPI) to detect intrusions, scan for viruses, and filter Internet content. However, the algorithm must still overcome some hurdles, including becoming efficient at multi-gigabit processing speeds and scaling to handle large volumes of signatures. Researchers in packet processing used to be most interested in *longest-prefix matching* in the routing table on Internet routers and *multi-field packet classification* in the packet header for firewalls and quality-of-service applications. However, DPI for various signatures is now of greater interest. Intrusion detection, virus scanning, content filtering, instant-messenger management and peer-to-peer identification all can use string matching for inspection. Much work has been done in both algorithm design and hardware implementation to accelerate the inspection, reduce pattern storage space, and efficiently handle regular expressions.

According to our survey of recent publication about string matching from IEEE Xplore (`ieeexplore.ieee.org`) and ACM digital library (`portal.acm.org/dl.cfm`) in Table 1.1 and 1.2, researchers formerly were more interested in

Table 1.1: Number of the IEEE/ACM publications for each application of string matching.

IEEE/ACM	2004 -	2000 - 2003	1990s	1980s	1970s
DPI	49	12	0	0	0
computational biology	13	16	15	0	0
information retrieval	24	19	26	0	0
pattern recognition	17	20	24	2	0
DBMS	4	11	1	0	0
compression	3	10	18	2	0
other app.	11	11	9	5	5
pure algorithm	31	39	110	22	12
total	152	138	203	31	17

Table 1.2: Number of the IEEE/ACM publications for each specific implementation method other than ordinary implementation.

IEEE/ACM	2004 -	2000 - 2003	1990s	1980s	1970s
ASIC/FPGA	34	9	9	2	0
network processors	3	0	0	0	0
multiple processors	10	3	10	2	0

pure algorithms for either theoretical interest or general applications, while algorithms for DPI have attracted more attention lately. Likewise, to meet the demand for higher processing speeds, researchers are focusing on hardware implementation in application-specific integrated circuits and field-programmable gate arrays, as well as parallel multiple processors.

In DPI, automaton, heuristic and filtering approaches are common. We leave out the bit parallelism approach and only offer pointers to it for completeness because it is often used in computational biology but rarely in networking. We assume the text length to be n characters and the pattern length (or the shortest length in case of multiple patterns) to be m characters. Other characteristics of

string-matching algorithms are listed in the following:

Characteristics of string-matching algorithms

Times of searches Some applications, such as search engines, search the same text many times for different querying strings. Building an indexing data structure from the text in advance is therefore worthwhile to perform indexed search with the time complexity as low as $O(m)$. In contrast, the applications in networking and biological sequences search throughout the text on-line only once without the indexing structure and the time complexity is linear in n.

Text compression Some algorithms directly search the compressed text with minimum (or no) decompression, while others scan over the plain text.

Matching criteria A match can be exact or approximate. The pattern and the matched piece of text should be identical in the former, while a limited number of differences between them is allowed in the latter.

Time complexity Some algorithms have deterministic time complexity of linear time, while others achieve sub-linear time by skipping characters not in a match. The latter may be faster on average, but not in the worst case.

Number of patterns An algorithm can scan for a single pattern or multiple patterns simultaneously.

Expressiveness in pattern specifications Pattern specifications range from fixed strings to regular expressions in various syntax options. Besides primitive notations of alternation, catenation and Kleene closure, extensions in the syntax of regular expressions include the UNIX representations, the ex-

tended forms in POSIX 1003.2, and Perl Compatible Regular Expression (PCRE) [Fro06]. An increasing number of signatures are specified in regular expressions for their expressiveness.

1.1.1 Automaton-based approach

An automaton-based approach tracks partially matched patterns in the text by state transition in either a deterministic or a non-deterministic finite automaton (DFA or NFA) that accepts the strings in the pattern set. A DFA implementation generally has lower time complexity but demands more space for pattern storage, while an NFA implementation is the opposite [YCD06]. The automaton-based approach is popular in DPI for two reasons: (1) The deterministic execution time guarantees the worst-case performance even when algorithmic attacks deliberately generate text to exploit an algorithm's worst-case scenario. (2) Building an automaton to accept regular expressions is systematic and well studied.

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

Reducing sparse transition tables A transition table is generally sparse because most states, particularly those away from the root state, have only few valid next states. The table can be compressed by storing only links

to valid next states after one or more input characters and failure links of each state. The state transition table, the failure links and the lists of matched patterns in the final states can be stored separately in a software implementation to improve the cache locality during tracking. Snort (`www.snort.org`), a popular open-source intrusion-detection package, has carefully tuned the data structure in this way to improve the cache performance. The latest revision uses a basic NFA construction as the default search method (*src/sfutil/bnfa_search.c* in the source tree of Snort 2.6.1).

Reducing transitions With the extended ASCII alphabet, an automaton has a maximum of 256 transitions from a state. Splitting an automaton into several smaller ones at the bit level can reduce the number of transitions. For example, suppose the automaton is split into eight, and then one automaton is fed with b_7, one is fed with b_6, and so on, where $b_7 b_6 \ldots b_0$ denotes the eight bits of the input characters.

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

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

of transitions for regular expressions on practical products and tools.

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

Rewriting and grouping Some combinations of wildcards and repetitions in regular expressions will generate a complex automaton that grows exponentially [YCD06]. It is possible to rewrite the regular expressions to simplify the automaton because we do not have to find every match in the text in some networking applications. Finding an appearance of certain signatures suffices. For example, every string s identified by "ab+" ("+" denotes one or more) can be identified by "ab" as s itself or a prefix of s, so reporting a match against "ab" is sufficient to report an appearance of "ab+".

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

Hardwiring regular expressions Some designs use building blocks on the FPGA to match various patterns. The implementation typically prefers an NFA

to a DFA because an NFA has fewer states, and the inherent concurrency of hardware can easily track multiple active states. A few techniques can reduce the area cost of building blocks. For example, identical substrings from different patterns can share common blocks. Specific hardware logics can directly handle notations in regular expressions such as class of characters, repetitions, wildcard characters and so on.

1.1.2 Heuristic-based approach

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

Shift values Because the positions or the shift values corresponding to possible blocks are computed and stored in a table beforehand, a table lookup drives shifting the search window in the search stage. Figure 1.1 illustrates a simple but generic heuristic for only one pattern to visualize why skipping is efficient. In the upper part, because "FGH" is not a substring of the pattern and its suffix is not a prefix of the pattern, shifting the search window by $m = 6$ characters without examining the remaining characters in the window will not miss a match. After the shift, "XYZ" becomes the suffix of both the pattern and the window, meaning a suspicious match occurs. The entire window is then verified, and a match is found.

However, if a suffix of the block is the prefix of some pattern, the shift value should be less than m because the suffix may be the prefix of that

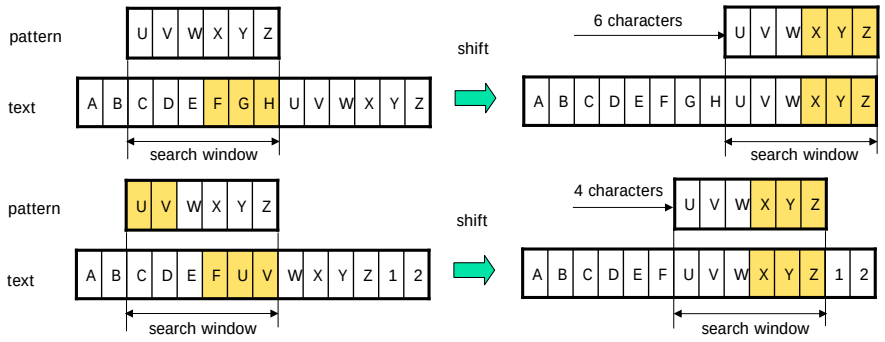


Figure 1.1: A simple heuristic in a heuristic-based approach.

pattern after the shift. Figure 1.1 illustrates this case. This heuristic can be easily extended to handle patterns shorter than the block size. If a short pattern is a substring of the block, looking up the block can claim a match. In addition to the heuristic for matching fixed strings, Navarro and Raffinot presented a heuristic to skip text characters for regular-expression matching [NR04].

Ideally, most shift values are equal or close to the pattern length m, so the time complexity is sublinear, i.e., $O(n/m)$. However, the time complexity could be $O(nm)$ in the worst case, in which each entire search window is examined after a shift of only one character. Although methods exist to guarantee the linear worst-case time complexity for a single fixed string or regular expression [NR04, Gal79], they are rarely adopted in DPI, which looks for multiple patterns.

Due to their vulnerability to algorithmic attacks, heuristic-based algorithms usually are not preferable for network-security applications because an attacker might manipulate the text to degrade performance. Because applications such as Snort have short patterns of only one or two characters, the small value of m makes the advantage of skipping marginal. Nev-

ertheless, for applications with long patterns such as the signatures of non-polymorphic viruses in the anti-virus package ClamAV (`http://www.clamav.net`), skipping over the text is still helpful.

Implementation details Block size, mapping from the blocks to derive shift values, and other implementation details can significantly affect practical performance. When choosing proper parameter values, considerations include the size of the pattern set, block distribution, cache locality and verification frequency. For example, a large block has fewer chances to appear in the patterns, resulting in less frequent verification. However, a large block also generally implies a large shift table, and so reduces the cache locality. Careful experimenting should properly tune these parameters. When suspicious matches frequently appear, implementing an efficient method to identify the matched pattern is also important.

Because the block distribution may be non-uniform in practice, some blocks may appear much more often than expected, shortening the shift distance and increasing the verification frequency. Checking the matches in additional blocks within the search window can reduce the verification frequency.

Using a heuristic similar to that in Figure 1.1 to look for the longest suffix of the search window that is also a substring of some pattern might result in long shift distance even with non-uniform block distribution. However, longer shift distance does not always imply better performance. The overheads due to the extra examination should be carefully evaluated.

1.1.3 Filtering-based approach

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

Text filtering A common method of filtering the text is using the Bloom filter, characterized by a bit vector and a set of k hash functions $h_1, h_2, \ldots, h_k$ mapped to that vector. When multiple patterns are present, the patterns of a specific length are stored in a separate Bloom filter by setting to 1 the bits the the patterns' hash values address. The search queries the set of Bloom filters by mapping the substrings in the text under inspection to them with the same set of hash functions. Specifically, a substring x under inspection is mapped to the Bloom filter storing the patterns of length $|x|$. If one of the bits in $h_1(x), h_2(x), \ldots, h_k(x)$ are not set to 1, x certainly is not in the pattern set; otherwise, x might be in the pattern set, and the match must be further verified. The uncertainty comes from different patterns setting checked bits. The false-positive rate is a function of the bit vector size, the number of patterns and the number of hash functions. Properly controlling these parameters can reduce the false-positive rate.

Parallel queries Parallel queries to the Bloom filters are generally implemented in hardware for efficiency, but efficient software implementation of sequential queries is also possible. For example, the implementation can sequentially query with a set of hash functions, from simple to complex ones, to

look for pattern prefixes of a certain length and verify a match if a prefix is found. The simple hash functions are designed to be rapidly computed and can filter most of the text, so the search is still fast. There might be many Bloom filters for a wide range of pattern lengths, because each length requires one. A solution is to limit the maximum pattern length allowed and break a long pattern into short ones. If all substrings of a long pattern appear contiguously and in order, that pattern is present.

The filtering-based approach does not directly support some notations in regular expressions such as wildcards and repetitions. An indirect solution is extracting the necessary substrings from the regular expressions, searching for them, and verifying the match if these substrings appear. For example, ClamAV divides the signatures of polymorphic viruses into ordered parts (i.e., substrings of the signatures), and tracks the orders and positions of these parts (with a variant of the Aho-Corasick algorithm) in the text to determine whether a signature occurs. Table 1.3 summarizes the key methods as well as the pros and cons of each.

1.2 Current trends in DPI

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

1.2.1 Matching expressive pattern specifications

Expressive pattern specifications, such as regular expressions, can accurately define the signatures. Efficient solutions to matching regular expressions in DPI are

Table 1.3: Summary of approaches to string matching for DPI.

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

therefore attracting considerable interest. Bispo et al. compared several designs for regular expression matching [BSC06]. Most of them were can perform regular expression matching on the order of several gigabits per second. Commercial products, including Cavium Octeon MIPS64 processor family (`www.cavium.com/OCTEON_MIPS64.html`), SafeNet Xcel 4850 (`www.safenet-inc.com/products/chips/safeXcel4850.asp`) and Tarari RegEx5 content processor (`www.lsi.com/documentation/networking/tarari_content_processors/Tarari_RegEx_Whitepaper.pdf`), all claim to support regular-expression matching at gigabit rates. String matching, a problem once believed to be a bottleneck, has become less critical

given the latest advances.

Most existing research aims at intrusion-detection applications, especially Snort, which has thousands of signatures, but antivirus applications such as ClamAV claim a signature set of more than 200,000 patterns to date. We believe a more scalable and efficient design for matching a huge set of expressive patterns deserves further study. Moreover, some patterns may belong to only a specific protocol, file type, etc., and some are significant only when they appear in specified positions of the text. Rather than assuming a simple model of searching for the whole pattern set throughout the entire text, a design can consider the additional information to optimize the performance. An efficient software implementation for these cases is also desired as hardware accelerators are not always affordable in practical applications.

1.2.2 Accelerating packet content processing

Although numerous research efforts have been dedicated to string matching, packet processing in DPI involves even more effort. Paxson et al. described the insufficiency of string matching in intrusion detection due to its stateless nature [PAD06], and envisioned a framework of architecture that attempts to exploit the parallelism in network analysis and intrusion detection for acceleration.

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

Commercial products are on this track. For example, the Cavium Octeon MIPS64 processor family involves a TCP unit, a compression/decompression en-

gine and 16 regular expression engines on a single chip, and claims the performance to be up to 5 Gbps for regular-expression matching plus compression/decompression.

1.2.3 Parsing content in high-level semantics

String matching in network applications may refer to contextual information parsed from high-level semantics [SP03]. For example, some patterns are significant only within the URIs. Spam and Web filtering also demand intelligence in high-level semantics to analyze the content, so does XML processing [GGM04]. String matching with high-level semantic extraction and analysis from the text is therefore beneficial.

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

1.3 Objective, methodology and road map of research

This objective of this dissertation is to accelerate string matching for DPI. We believe addressing all the aforementioned issues still has way to go, and the efforts in this dissertation is on the right track toward this goal.

After reviewing existing solutions, to better understand the characteristics of existing string-matching algorithms and their performance on practical appli-

cations, we implemented them in open-source packages and profiled their performance to know which algorithm is suitable for each application, as well as the key factors that affect the performance. Following the profile, we devised a hardware architecture to accelerate string matching for a large pattern set. The architecture features a solution to exploiting algorithmic heuristics in hardware and realizing string matching in sub-linear time. Some practice issues that might lead to the worst case are also addressed. However, only hardware solutions are insufficient because they may not be applicable in some situations. Therefore, we designed a hybrid algorithm for anti-virus applications. The algorithm splits the patterns into long and short ones, to exploit algorithmic heuristics for long patterns, while keeping good cache locality for short patterns. We also presented an early decision algorithm to accelerate Web filtering beyond string matching, since solely accelerating string matching is not enough in Web filtering.

The road map of the dissertation is as follows. Chapter 2 presents the profiling of each string-matching algorithm on DPI applications. The key factors to the performance are also discussed. Chapter 3 presents the hardware architecture for string matching in sub-linear time with algorithmic heuristics. Chapter 4 presents the hybrid algorithm for virus scanning in software. Chapter 5 presents the early decision algorithm for accelerating Web filtering. The conclusion and future work are given in Chapter 6.

CHAPTER 2

Profiling and Accelerating String Matching in Three Network Content Security Applications

2.1 Introduction

Detecting and filtering intrusions, worms, viruses and inappropriate Web pages involve string matching for designated signatures in the packet content, as opposed to packet classification that matches fixed fields in the packet header [GM01]. The position and length of the signatures are unknown beforehand, so scanning the packet payload for signatures is less efficient than packet classification. String matching was reported to be a bottleneck for these applications [FV01, LJL06], so its efficiency is critical.

Signatures in content security applications are represented as patterns in some forms. For clarifying the terminology, a string is a sequence of characters, and a pattern is an occurrence of a string in the text [NR02]. Signature characteristics in different applications may vary wildly in the number, length and character distribution in the alphabet. For instance, anti-virus systems feature a large number of long signatures, while intrusion detection systems may have short signatures of one or two characters. No existing string matching algorithms can search for signatures of various characteristics faster than others can, so choosing a proper algorithm becomes important.

This work reviews existing string matching algorithms and their applications in network content security. The characteristics of signatures in three typical open source packages are investigated: ClamAV (`www.clamav.net`) for anti-virus, DansGuardian (`dansguadian.org`) for Web filtering and Snort (`www.snort.org`) for intrusion detection systems (IDS). This work profiles the performance of various algorithms, identifying and implementing the fastest algorithm for each package. The improved packages are then benchmarked for sample sets of both synthetic and real data. The impact of memory and cache accesses on performance is also measured quantitatively. This work also proposes classified RKBT (Rabin-Karp with binary search and two-level hashing) to accelerate the RKBT algorithm [MM96], which can verify a possible match [KST03], and its efficiency becomes critical as the number of possible matches increases.

2.2 Related works

2.2.1 Categories of string matching algorithms

Single string matching searches the text $T = t_1 t_2 \dots t_n$ for all the occurrences of a string p, called the pattern, where n is the text length. Multiple string matching extends to search text for the pattern set $P = \{p_1, p_2 \dots, p_r\}$ simultaneously. Exact matching stipulates that the pattern and the matched text should be exactly the same, while an approximate matching algorithm allows a limited number of errors between a pattern and the matched text. This work focuses on only the former because the majority of content security applications use exact matching.

Exact string matching algorithms can be categorized in various ways, one of which is grouping them into three general approaches: prefix searching, suffix searching and factor searching, depending on which part of the pattern is searched

for within the search window [NR02]. A string X is the prefix, suffix and factor of XY, YX and YXZ, respectively, where Y and Z are also strings. The time complexity of an algorithm can be linear or sub-linear. The latter is feasible by skipping characters that do not need to be examined in the text.

This work categorizes the algorithms to emphasize the data structure. The categories include automaton, heuristics, filtering and bit-parallelism approaches. An automaton-based algorithm builds a finite state automaton from the patterns and tracks in the text the partial match of the pattern prefixes by state transition. A heuristics-based algorithm skips the characters to accelerate the search according to certain heuristics, and a verification algorithm follows a possible match to verify if a true match occurs. A filtering-based algorithm looks for characteristics of the patterns in the text to see whether or not a possible match occurs, and also verifies it for a true match. A bit-parallelism-based algorithm simulates the operation of a non-deterministic finite automaton that tracks the partial match of the prefix or the factor of the pattern by means of the parallel bit operations inside a computer register word in which the state transition status is encoded [NR00]. Chapter 1 of this dissertation reviews common algorithms for network content security applications.

2.2.2 Selected packages and their algorithms

2.2.2.1 ClamAV

ClamAV contains two types of virus patterns: basic patterns that are a simple sequence of characters, and multi-part patterns composed of multiple sub-patterns. ClamAV scans basic patterns by the Wu-Manber algorithm. If no virus is found, a variant of the Aho-Corasick algorithm then scans for the multi-part patterns, in which the automaton is represented as a two-level trie [MDW04]. The sub-

patterns with common two-character prefixes are stored in a linked list under the leaf that represents the common prefix. ClamAV uses a table to keep the sub-patterns and their positions that have been found for each pattern. All sub-patterns of a multi-part pattern must be matched in sequence to assert a virus. ClamAV also supports a simplified form of regular expressions. For example, ClamAV allows "bounded gaps" that specify the minimum and maximum distances allowed between two consecutive sub-patterns. Recording the position of a sub-pattern and calculating the distance from its last sub-pattern can verify whether the bounded gaps are satisfied.

2.2.2.2 DansGuardian

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

the Horspool algorithm will search for all the keywords one by one.

2.2.2.3 Snort

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

2.3 The Verification algorithm

Some algorithms such as the BG and WM algorithms rely on a verification algorithm to assert a true match when a possible match is found. The RKBT algorithm can serve this purpose [KST03], but it is inefficient for a large pat-

tern set due to its data structure discussed below. We propose Classified RKBT (CRKBT) to accelerate the verification.

2.3.1 The CRKBT algorithm

Figure 2.1 (left) illustrates the operation of the RKBT algorithm. Each pattern is viewed as consecutive blocks of four bytes, so each block can form a 32-bit integer. If the pattern length is not a multiple of four, the last block is padded with zeros. The first hash function is defined by xor'ing the integers in these blocks. In the preprocessing, an ordered table stores the first 32-bit hash values of the patterns. A second hash function is derived by xor'ing the lower and the upper 16 bits of the first hash values. A bitmap of 2^{16} entries is then built to store the second hash values. The i'th bit of the bitmap is 1 if at least one pattern has i as its second hash value, and is 0 otherwise. The bitmap indicates whether binary search in the ordered table is necessary. If the the second hash value of a search window in the bitmap is 0, no possible match will occur and the verification fails; otherwise, say the 2345th bit in Figure 2.1, the ordered table is looked up in the first hash value with binary search. If the first hash value is found, the patterns with that value are compared with the search window to check if a true match occurs; otherwise, the verification fails.

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

We propose a classified variant, namely CRKBT, to accelerate RKBT. Figure 2.1 (right) illustrates its operation. CRKBT divides the ordered table for

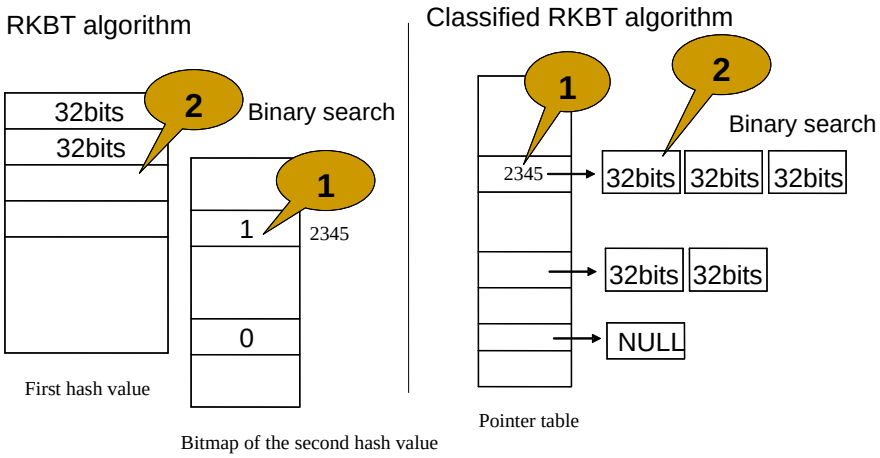


Figure 2.1: The operations of the RKBT algorithm (left) and the Classified RKBT algorithm (right).

binary search into several small tables associated with the second hash values. The search scope is reduced to only a subset of the patterns that have the same second hash value, so the binary search is much faster. A pointer table replaces the bitmap in RKBT. The i'th pointer points to an ordered table of only patterns that have the second hash value i, and points to NULL if there is none. The pointer table is looked up in verification. If the pointer of the second hash value is not NULL, the ordered table that the pointer points to is searched with binary search. The overhead of the CRKBT algorithm is 256 KB of the pointer table (2^{16} entries * 4-byte pointer) and at most 256 KB (4-byte integer for the length) to store the sizes of the divided ordered table.

2.3.2 Experiments on the CRKBT algorithm

The execution time of both the RKBT and CRKBT algorithms is benchmarked as follows. The text of 32 MB and the patterns are randomly generated from the alphabet of 8-bit characters. The shortest pattern length is 8. The tests run on

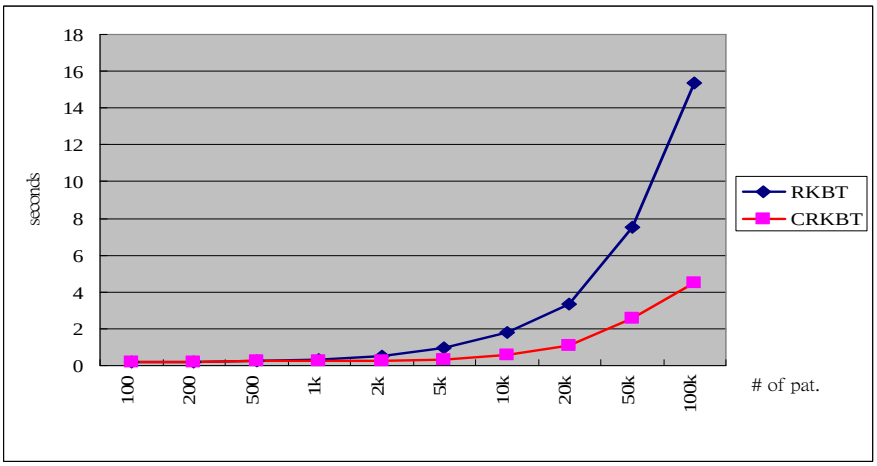


Figure 2.2: The execution time of the RKBT and CRKBT algorithm.

a computer with a 2.8 GHz Pentium 4 processor, 1 GB of memory and 512 KB L2 cache. Both algorithms are implemented in C, and run on Linux kernel 2.6.5. Figure 2.2 presents the benchmark results.

The execution time of both algorithms for small pattern sets is close because of few possible matches and thus few chances of binary search in the ordered table. The number of possible matches and thus the difference in the execution time increase with the number of patterns. The scope for binary search is small in the CRKBT algorithm (only the subset of patterns with the same second hash values), so the binary search is fast. CRKBT is four times faster than RKBT for a pattern set of 100,000 patterns, so it is more scalable to a large pattern set.

2.3.3 Analysis of the CRKBT algorithm

Binary search in a large pattern set can dominate the verification time. The CRKBT algorithm reduces the ordered table size by dividing the patterns into subsets. Assume the number of patterns is r. Both algorithms check the bitmap or the pointer table first. In RKBT, the probability that a bit is set to 1 is p,

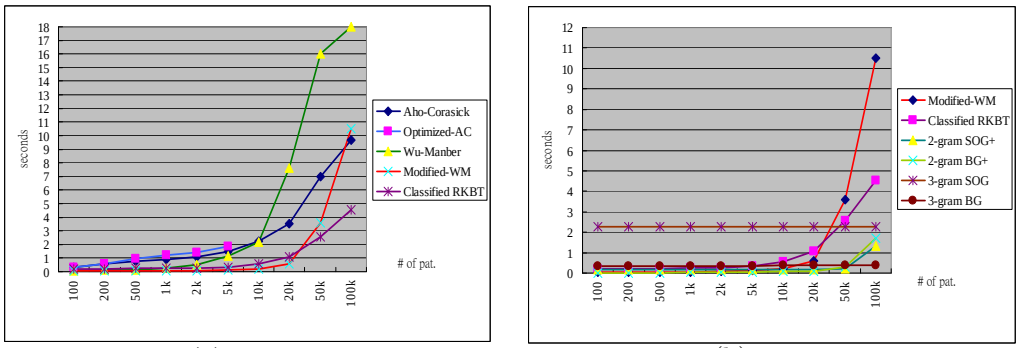
where p is estimated to be $1-(\frac{65535}{65536})^r$. If the bit is set to 1, binary search in the ordered table follows, and the expected number of memory accesses in the table is $\log_2 r + 1$. Otherwise, the verification fails. Therefore, the expected number of memory accesses in the RKBT algorithm is $p(\log_2 r + 1) + (1 - p)$. Because the expected size of each ordered table to which a pointer points in CRKBT is $1 + \frac{1}{65536}$, the expected number of memory accesses in the binary search becomes only $\log_2(1 + \frac{1}{65536}) + 1$, which is much smaller than $\log_2 r + 1$ in RKBT for a large r. The expected number of memory accesses in the binary search of the CRKBT algorithm is then $p(\log_2(1+\frac{1}{65536})+1)+(1-p)$. Therefore, the CRKBT algorithm is more scalable than the RKBT algorithm.

2.4 Profiling algorithms of string matching algorithms

In the external and internal profiling, the benchmarking environment and configuration are the same as those described in Section 2.3.2. The algorithms involved in the profiling include the Wu-Manber algorithm [WM94], Aho-Corasick algorithm [AC75], the SOG and BG algorithms [KST03], and some variants of them (including Modified-WM and Optimized AC in Snort [NRa]). We also use CRKBT as the verification algorithms of SOG and BG, and denote both revised version by SOG+ and BG+. These algorithms are profiled for various pattern lengths and pattern set sizes.

2.4.1 External profiling

The external profiling measures the execution time of scanning text of 32 MB. Figure 2.3 presents the benchmark results for LSP = 8 first, where LSP denotes the length of the shortest pattern. The benchmark results for LSP $<$ 8 will be



(a) part 1 (b) part 2

Figure 2.3: Comparison of execution time of selected string matching algorithms.

discussed in Section 2.4.3.

Figure 2.3(a) presents that the Modified-WM algorithm is the fastest for the pattern set size smaller than 20,000. When the pattern set is larger, the CRKBT algorithm is the fastest. The execution time of the Optimized AC for the pattern set size larger than 5,000 is not presented because the execution takes too long to stop. The problem might be due to a bug in the Snort implementation. Figure 2.3(b) compares the execution time of the Modified-WM and CRKBT algorithms with that of the BG+ and SOG+ algorithms. The 2-gram BG+ algorithm is the fastest for the pattern set size smaller than 50,000. For a larger pattern set, the 3-gram BG+ algorithm is the fastest.

2.4.2 Internal profiling

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

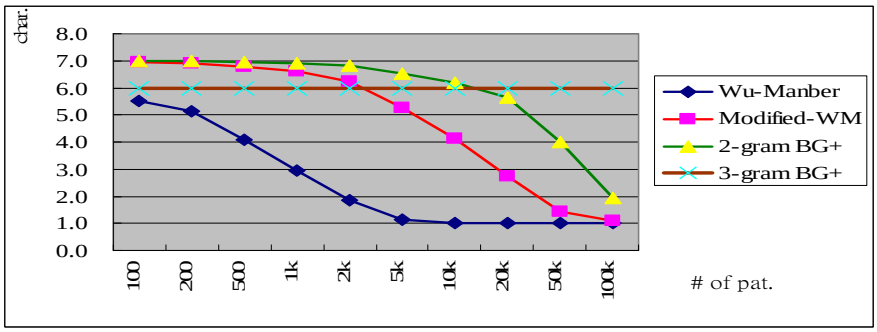


Figure 2.4: Comparison of the profiling results of average shift distance.

2.4.2.1 Shift distance

Both the WM and BG+ algorithms can skip certain characters in the search window. Figure 2.4 presents their average shift distance. The average shift distance of the WM algorithm is close to one character for the pattern set size between 5,000 and 100,000, so this algorithm can barely skip a character in this case. The average shift distance of the Modified-WM algorithm is greater than that of the WM algorithm, which can explain why the Modified-WM algorithm is faster. The results also explain why the 2-gram BG+ algorithm is the fastest for a small pattern set, and why the 3-gram BG+ algorithm is the fastest for a large pattern set due to the long average shift distance.

2.4.2.2 The percentage of possible matches

Some algorithms use a filtering approach, and verification may dominate the execution as the number of possible matches increases. Figure 2.5 shows the percentage of possible matches for each algorithm. The Modified-WM algorithm has fewer possible matches than the WM algorithm, so it is faster. The fast increase of possible matches in the WM algorithm indicates that it is not scalable to a large pattern set. The figure also explains why the BG+ algorithm is faster

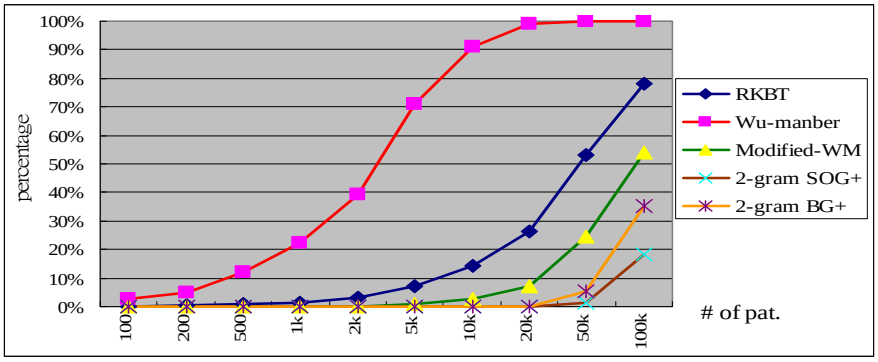


Figure 2.5: Comparison of the percentage of possible matches for each algorithm.

than the Modified-WM algorithm because of its fewer possible matches.

2.4.2.3 Memory accesses

It is insufficient to justify the external profiling results solely from the shift distance and the percentage of possible matches. For example, why does the CRKBT algorithm is the fastest in Figure 2.3(a) for the pattern set size larger than 50,000? Figure 2.6(a)-2.6(c) each presents the number of memory accesses of the algorithms in the same category (categorized in Section 2.2.1) profiled by Valgrind (`valgrind.org`) to observe the reason. For algorithms in the same category, the fewer the memory accesses, the faster the algorithm in the profiling. However, it is not the case for algorithms in different categories, as presented in Figure 2.6(d). For instance, the CRKBT algorithm has more memory accesses than the AC algorithm, but the former is faster. The number of memory accesses is still insufficient to justify the results.

More memory accesses do not imply longer time spent in accessing the memory due to cache locality. Figure 2.7 presents the cache misses for the CRKBT algorithm are fewer than those for the Modified-WM and AC algorithms. The number of cache misses for the 2-gram BG+ algorithm is the least. We can jus-

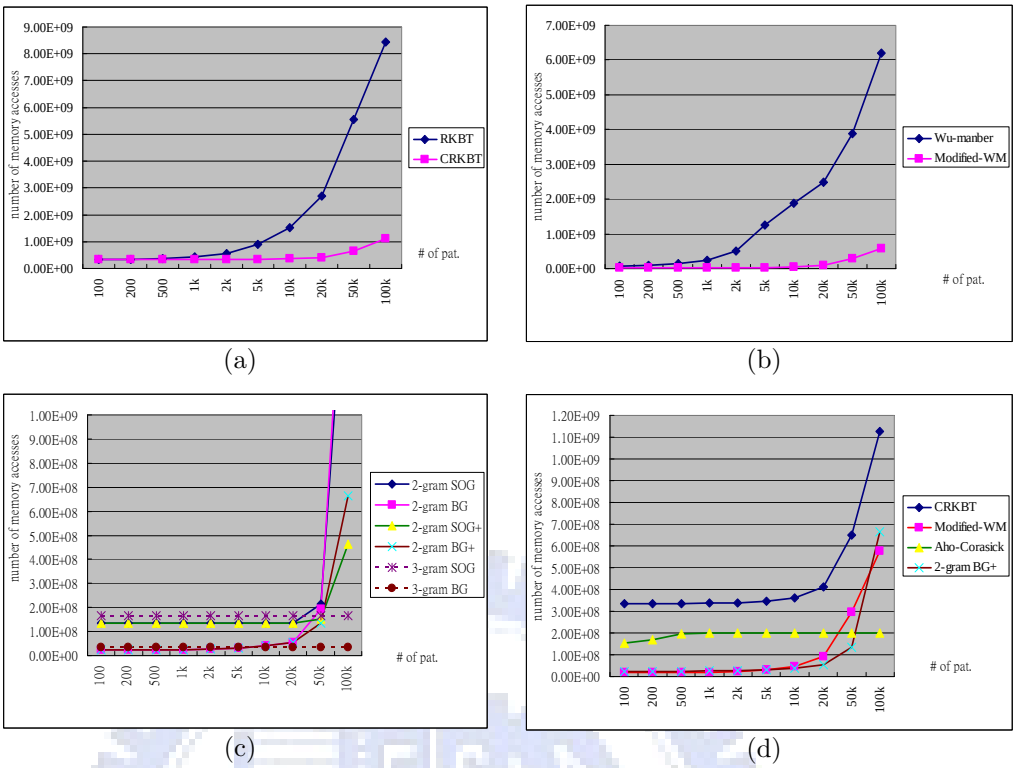


Figure 2.6: Comparison of the number of memory accesses in the each algorithm.

tify the prior results from the cache misses, including that the CRKBT algorithm is faster than the Modified-MW and AC algorithms for a large pattern set. In addition, the efficiency of the 2-gram BG+ algorithm is also justified.

In addition to the number of memory accesses, the memory consumption of each algorithm is also observed in Figure 2.8. The CRKBT and 2-gram BG+ algorithms have slow growth in memory consumption, primarily due to the increasing ordered table sizes for binary search. The Modified-WM algorithm uses fixed memory size to build the shift table and hash table for verification. The memory consumption in both the AC and Optimized-AC algorithms grows faster than that in the others as the pattern set increases. The memory consumption of the Optimized AC algorithm for the pattern set size larger than 5,000 is not presented because a possible bug impedes the correct execution.

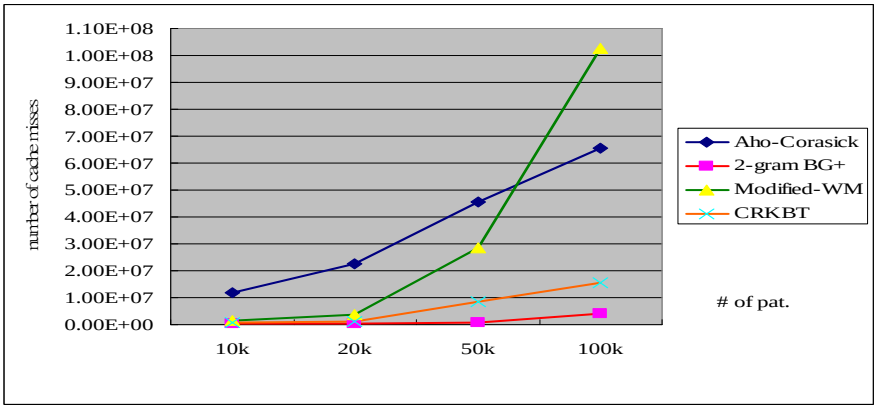


Figure 2.7: Comparison of the number of cache misses in each algorithm.

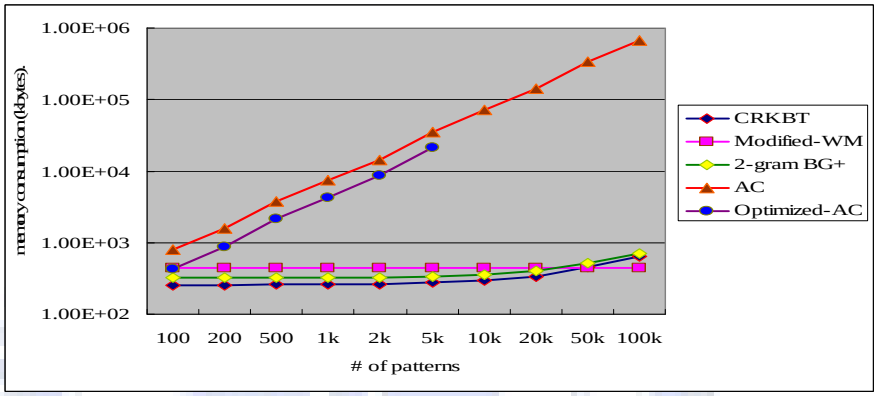
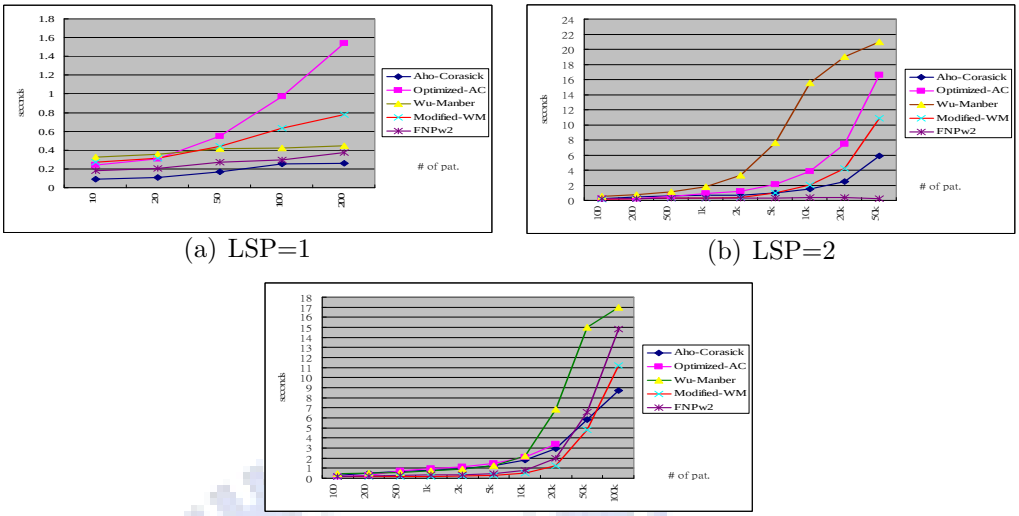


Figure 2.8: Comparison of the memory consumption in each algorithm.

2.4.3 Profiling for short patterns and summary

The external and internal profiling presents that the 2-gram BG+ algorithm is the fastest for LSP = 8 for the pattern set size smaller than 50,000, and the 3-gram BG+ algorithm is the fastest for LSP = 8 for a larger pattern set size. We also profile the performance for the LSP between 1 and 7. The ranks of each algorithm in efficiency for LSP between 4 and 7 are similar to that for LSP = 8, so the results are not presented. However, the ranks for LSP between 1 and 3 differ. Figure 2.9 shows the AC, FNPw2 and Modified-WM algorithms are the

fastest for $LSP = 1, 2$ and 3, respectively. Figure 2.10 summarizes the fastest algorithm for various pattern set sizes and pattern lengths.



(a) LSP=1 (b) LSP=2

(c) LSP=3

Figure 2.9: Comparison of the execution time for LSP = 1, 2 and 3. (FNPw2 denotes FNP with $w = 2$.)

2.5 Experiments on real applications

2.5.1 Implementation in three content security packages

Each application has different pattern lengths and pattern set size. The shadow area in Figure 2.10 also indicates the range of pattern lengths and pattern set size of each application, overlapping with the profiling results. The shaded arrows indicate an application has patterns longer than the lengths in the the shadow area. This figure suggests which algorithm be better implemented in each application. We revised the packages in Section 2.2.2 by implementing the suggested algorithms and observed the acceleration below.

ClamAV The LSP of basic patterns in ClamAV is 10 and the number of patterns

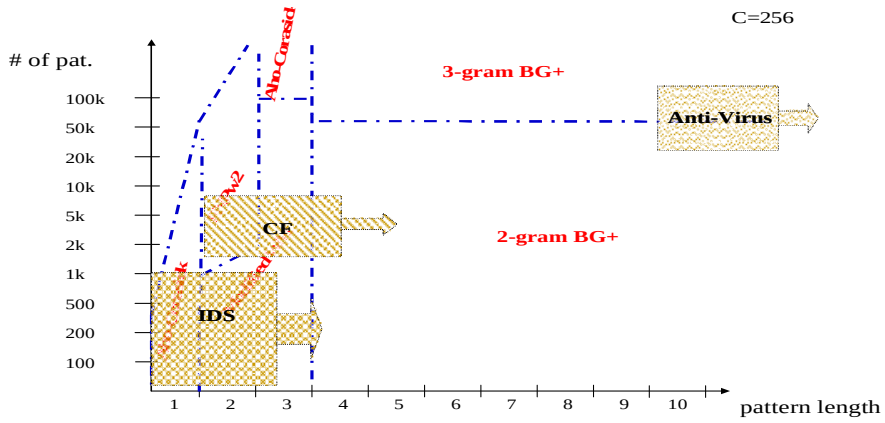


Figure 2.10: The profiling summary of each algorithm. (C denotes $|\Sigma|$.)

is larger than 30,000 to date. We replace the WM algorithm with the 2-gram BG+ algorithm to match basic patterns. When the pattern set is even larger in the future, the 3-gram BG+ algorithm can be used to enhance the efficiency. The AC algorithm is still responsible for matching regular expressions of multi-part patterns.

DansGuardian According to our investigation, the Horspool algorithm scans 25 content keywords one by one in the current implementation. If the *forcequicksearch* option is enabled, every pattern in the pattern set will all be searched for with the Horspool algorithm. We do not enable this option because enabling this option will have the text scanned as many times as the number of patterns, and will actually slow down the search. We group all the patterns together and implement the Modified-WM algorithm to handle short patterns with LSP = 2 and 3, and the 2-gram BG+ algorithm to handle the longer patterns.

Snort Snort groups patterns into rule sets according to the packet header. The LSP of every rule set is not the same. We implement a hybrid method instead of enabling the default method, the Modified-WM algorithm. The

AC algorithm is selected for LSP=1; otherwise, the Modified-WM algorithm still handles the pattern matching.

2.5.2 Benchmarking of the revised implementation

The speedup of the revised packages are benchmarked in this section. The performance for both the real and synthetic sample data is also compared, where the synthetic data are generated from uniformly distributed random characters. The comparison of both types of sample data can exhibit whether the observation for the synthetic data is also applied to real situations.

2.5.2.1 Benchmarking for ClamAV

We select 10 Windows execution files whose sizes are between 32 KB and 16 MB as real data in the benchmark, which also tests for synthetic data of the same size. Figure 2.11 compares the execution time of both the original ClamAV and its revised version. The difference in scanning time between both versions becomes obvious with increasing file size. For example, the revision is five times faster than the original one when the file size is 16 MB. The acceleration comes primarily from that reduction of verification during the search. Figure 2.11 also compares the execution time for real and synthetic data in both versions. The difference between both data types is almost unnoticeable because the character distribution in the patterns and files is close to random in ClamAV.

2.5.2.2 Benchmarking for DansGuardian

We use wget (`www.gnu.org/software/wget/wget.html`) to mirror an RFC Web site at `asg.web.cmu.edu/rfc/rfc-index.html` that contains more than 8,000

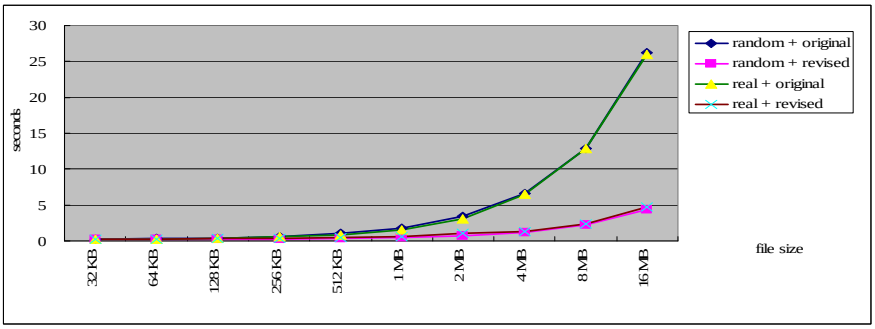


Figure 2.11: The performance improvement for both random and real data in the revised version of ClamAV.

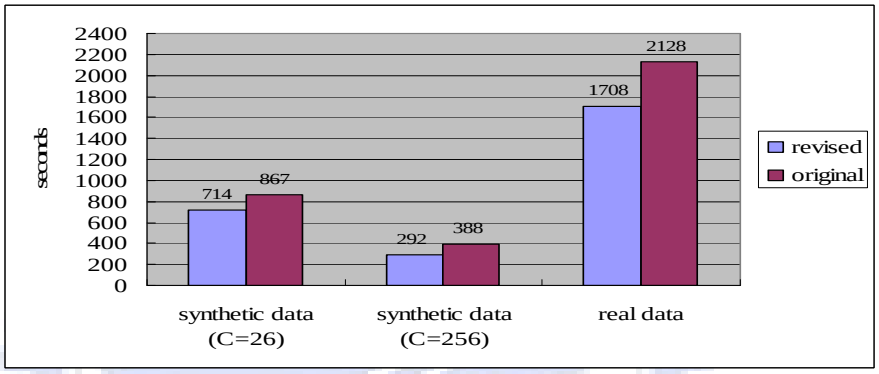


Figure 2.12: The performance improvement for both random and real data in the revised version of DansGuardian. (C denotes $|\Sigma|$.)

files, including HTML files and ordinary text files. DansGuardian's content filtering function scans these files. Figure 2.12 shows the original implementation takes 2128 seconds to mirror the entire site while the revised implementation takes 1708 seconds. The acceleration is insignificant because the verification algorithm has to find every possible match that has the same hash value. The filtering part in the searching process becomes less significant, and so is its acceleration.

We also generate synthetic Web pages of the same sizes for comparison with the real Web pages. First, we generate data from the character set of 256 char-

acters. The execution for synthetic data is faster than that for real data, because the character distribution of synthetic data is close to uniform but that of real data is biased towards English character set. Because the characters in real data concentrate more on English characters, the character set is effectively to be a small one. More possible matches occur and more verification is required than those for a uniformly distributed character set.

The character set of only 26 characters is also tested. The number of possible matches increases, and the processing time of content inspection is three times longer than that in the last experiment. However, the speed in this case is still much faster than that for the real data because keywords are more likely to appear in real data than in randomly generated synthetic data, so more possible matches occur and more verification is required.

2.5.2.3 Benchmarking for Snort

The HTTP traffic accounts for a large quantity of the Internet traffic, so we feed HTTP traffic to Snort. Snort is configured to run in the inline mode to easily measure its throughput. Figure 2.13 presents the benchmarking result of the throughput. First, we use a single client to mirror the entire site, but the acceleration is insignificant. We then add up to five clients for more traffic, and the acceleration becomes a little more obvious. However, the enhancement is still insignificant because Snort inspects only the HTTP header instead of HTTP body in most cases [NRb], and only a small portion of the traffic is inspected.

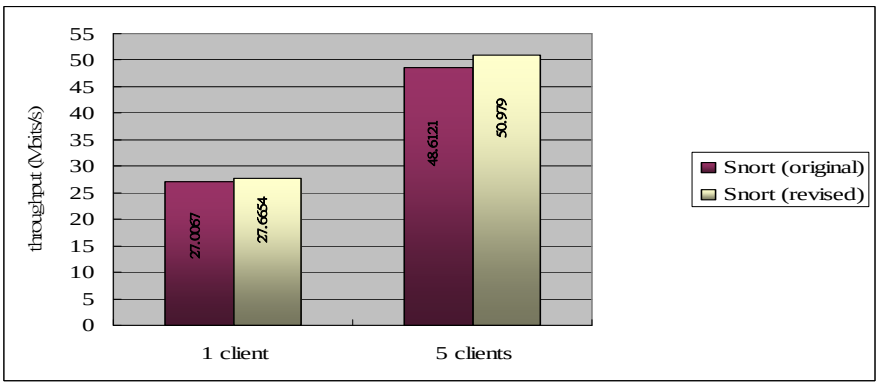


Figure 2.13: The benchmarking result of Snort.

2.6 Conclusion

This study reviews and profiles typical string matching algorithms to observe their performance under various conditions and sheds light on choosing the most efficient algorithm for network content security applications. The AC algorithm is suitable for LSP=1. For LSP=2, the Modified-WM algorithm is suitable when the pattern set size is smaller than 1,000, and the FNPw2 algorithm is suitable for a larger pattern set. The Modified-WM algorithm is suitable for LSP=3, and the BG+ algorithm is suitable for LSP$\geq$4. Implementing the algorithms into real applications also justifies the results by improving the performance. The results also help to select an efficient algorithm for future applications.

String matching on intrusion detection is not so critical in the profiling because the detection should scan only the content for only the patterns that are significant. Therefore, we believe accelerating the entire process, not only string matching, should deserve further study in the near future.

CHAPTER 3

Realizing a Sub-linear Time String-Matching Algorithm with a Hardware Accelerator Using Bloom Filters

3.1 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 [NR02]. 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-giga bits per second.

Many FPGA accelerators hardwire signatures into logic cells [CNM02, SP04], 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 re-programming time. Implementing the designs in ASIC is infeasible since an ASIC chip is not reconfigurable. Storing signatures in the memory simplifies the updating [ACF05, BP05]. 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 [AC75]. Some designs can accelerate the process by tracking multiple characters at once [SIH04, DL05, BCT06], 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 [DKS04, PP05, SPW05]. 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 [LLL06]. 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 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 non-uniform 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 Time (BFAST) algorithm. This architecture uses a set of Bloom filters [Blo70], 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 [BM77], 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 work is organized as follows. Section 3.2 reviews typical string matching algorithms and hardware accelerators. Section 3.3 presents the architecture of the BFAST algorithm. Section 3.4 presents the detailed hardware implementation. Section 3.5 evaluates this architecture and compares it with existing works. Section 3.6 concludes this work.

3.2 Existing Works and Literature Background

A multiple-string matching algorithm searches the text $\mathscr{T} = t_1 t_2 \dots t_n$ for occurrences of the patterns in a pattern set $\mathscr{P} = \{P_1, P_2, \dots, P_r\}$ 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 3.1 summarizes the notations in this paper.

Table 3.1: 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.
$P_i[j \ldots k]$	A substring from the j-th to the k-th character of P_i.
Σ	The character set. $\lvert\Sigma\rvert = 256$ 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.

3.2.1 String Matching Algorithms

The Aho-Corasick (AC) algorithm [AC75] 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 [Tar06, Cav05] and deterministic execution time for robustness to algorithmic attacks. The transition table of an automaton is compressed to reduce the memory requirement [TSC04, Nor04]. 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 [SIH04,DL05], but they have two drawbacks. (1) Compressing the transition table may need tricky techniques, if feasible, as the table grows with a large block. (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 [DL05].

The Boyer-Moore (BM) algorithm is the first that can skip characters not in a match based on algorithmic heuristics [BM77], which are illustrated in [BMI]. 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 pre-processing and scanning. The search window of m characters slides along the text during scanning according to the heuristics: if the rightmost 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 rightmost 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 rightmost 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 [Blo70]. For each pattern X, the filter sets to 1 the bits addressed by the k hash values $h_1(X)$, $h_2(X)$, ..., $h_k(X)$ 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.

3.2.2 Hardware Accelerators

String-matching hardware accelerators either hardwire the patterns into logic cells on 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 [Xil04]. 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 [MLL03], and

the throughput is up to 1.184 Gbps. Cho et. al. designed a pipelining architecture of discrete comparators [CNM02]. 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 [SP03], and the throughput can be up to 11 Gbps at 344 MHz, but its area cost is still high. Several following studies were devoted to area reduction, such as [SP04].

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 [TS06, Lun06, TLLar, LTLar, LTH07, TLL05], a filtering search window [DKS04, PP05, SPW05, AC07], or both [DL05]. 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.

3.3 The BFAST architecture

3.3.1 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 rightmost 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 [WM94].

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.

3.3.2 Deriving shift distance using Bloom filters

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

$$Pre_\tau(P_i) = \begin{cases} P_i[1 \dots \tau] & \text{if } \tau < |P_i|, \\ P_i & \text{otherwise.} \end{cases} \tag{3.1}$$

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

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].

[1]We noticed Liu et. al [LHC04] had a similar observation, but their heuristic based on the

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 rightmost occurrence of B_0 ends at position j of some pattern, the shift value is $m - j$. The *bad-block* heuristic depicted in Section 3.3.3 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.

Figure 3.1 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 = `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.

---

prefix rather than the *suffix* of the search window may skip over and overlook a suspicious match.

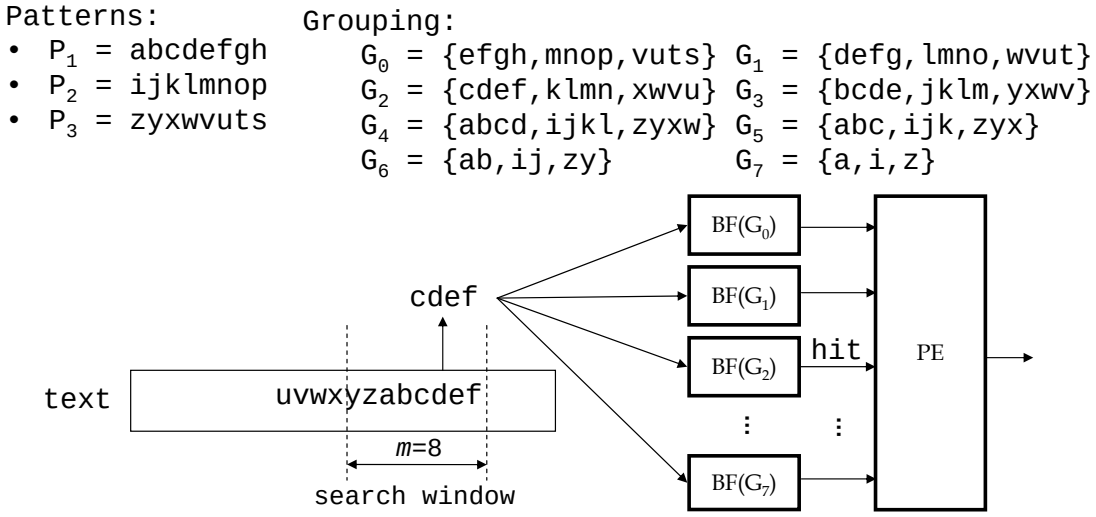


Figure 3.1: The blocks in the patterns are grouped for deriving the shift value from querying Bloom filters in parallel. If a block is a member of some group G_j, $BF(G_j)$ must report a hit. The priority encoder (PE) determines the shift value.

In general, when $m \geq b$, the groups are defined by

$$G_j = \begin{cases} \{P_i[m-j-b+1 \ldots m-j] \mid P_i \in \mathscr{P}'\}, \\ \qquad \text{if } 0 \leq j \leq m-b, \\ \{P_i[1 \ldots m-j] \mid P_i \in \mathscr{P}'\}, \\ \qquad \text{if } m-b+1 \leq j \leq m-1. \end{cases} \tag{3.2}$$

The queries check in parallel whether B_0 is a member of $G_0 \ldots G_{m-b}$ and whether the k-character suffix of B_0 is a member of G_{m-k}, for $k = 1 \ldots b-1$. 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

$$G_j = \begin{cases} \mathscr{P}', & \text{if } j = 0, \\ \{Pre_{b-j}(P_i) | P_i \in \mathscr{P}' \text{ and } |P_i| > b - j\}, & \\ \text{if } 1 \leq j \leq b-1. & \end{cases} \tag{3.3}$$

The patterns in G_0 are further divided into $G_0^{(1)} \ldots G_0^{(b)}$, 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, $s = min\{j|\ BF(G_j)$ 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.

3.3.3 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 '0x00 0x00 0x00 0x00' alone occupies 4.46% of the total blocks. If the suffix of some $P_i \in \mathscr{P}'$ 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,...,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 $\mathscr{H}$ be $\{i|BF(G_i)$ reports a hit and $i \geq j\}$. (1) If $\mathscr{H} \neq \emptyset$, the shift value derived by checking B_j is $i' - j$, where i' is the smallest value in $\mathscr{H}$. (2) Otherwise, the shift value is $m - j$. In (1), if $i' = j$, more checks may be needed as described below.

   Theorem 1. *The shift value derived here is safe.*

   Proof. Suppose a match occurs when the search window is shifted by a shorter distance, meaning 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 3.3.2 is a special case for $j = 0$. □

   Implementing the rules is simple. The priority encoder just ignores the report from $BF(G_0) \ldots BF(G_{j-1})$ 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.

2. 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

$$\frac{\lfloor E_{j+1}[s] \rfloor}{j+2} > \frac{max_j(s)}{j+1} \quad \text{or} \quad max_j(s) = 0 \tag{3.4}$$

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 pre-computed for each j according to the analysis in Section 3.3.6, 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 $\frac{\lfloor E_{j+1}[s] \rfloor}{j+2}$ might be zero due to integer division, even if $\lfloor E_{j+1}[s] \rfloor > 0$. The inequality may fail even though $max_j(s) = 0$.

3. 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.

Figure 3.2 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.

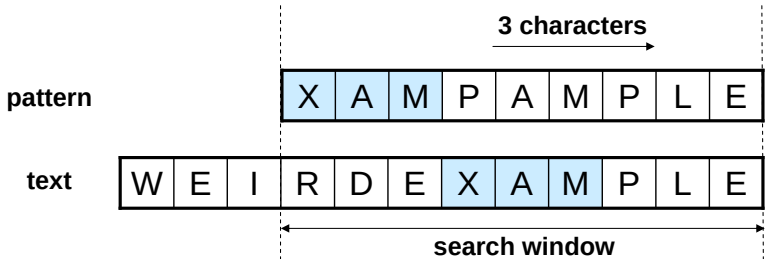


Figure 3.2: An illustration of the bad-block heuristic.

3.3.4 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 rightmost 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 3.3.2. 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 [Gal79]. 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 [Gal79, NR04], 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 $\frac{m\eta}{2}$, 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 5.1 of [NR04], 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 (1) find the longest prefix of some pattern in the window suffix, or (2) go back to the initial state of the automaton (if a failure occurs). In case (1), 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 Figure 3.3(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 Figure 3.3(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 Figure 3.3(a) or bad-block heuristics in Figure 3.3(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.

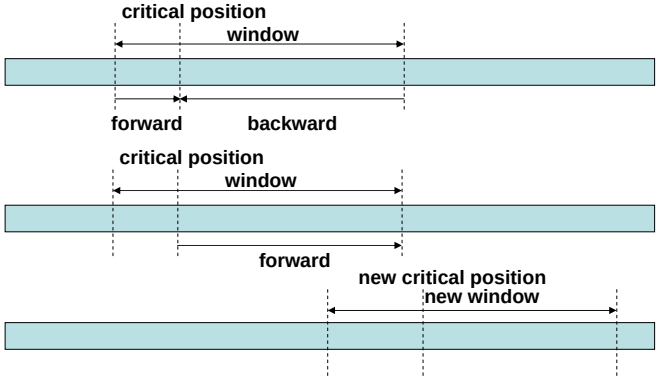
The shift thus will not miss a match. In the latter, we proved the shift will not miss a match in Section 3.3.3.

Linear time In each shift, the forward scanning is resumed either from or after the critical position where it is stopped last time (See Figure 3.3), 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 [YCD06]. 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 $O(n)$. □

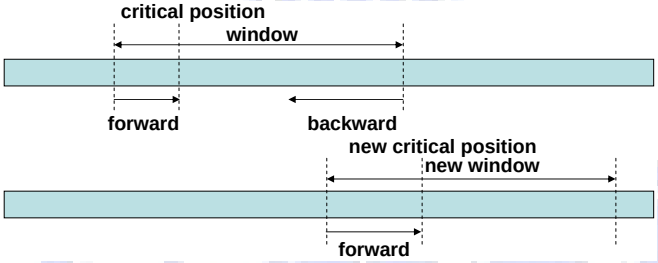
Although the second alternative can guarantee linear worst-case time and offer sub-linear time performance on average, it has two overheads. First, it needs the space to store the AC automaton for forward scanning. Second, it needs forwarding scanning in each shift to exclude blocks from being revisited 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 3.4.

3.3.5 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 $\mathscr{X}$ =



(a) Backward scanning reaches the critical position. Forward scanning starts from the critical position to the end of the search window, which is then aligned to the found pattern prefix or to begin at the position where the failure occurs.



(b) Backward scanning gets a shift value before reaching the critical position. The search window is shifted accordingly, and then the forward scanning tracks the overlapping part of the new window and the last window.

Figure 3.3: The procedure to maintain the linear time performance.

$\{0, 1, \ldots, 2^c - 1\}$ be a set of key values in c bits and $\mathscr{V} = \{0, 1, \ldots, 2^\nu - 1\}$ be addresses of a bit vector of ν bits. It is presented in [RFB97] 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 $< x_1, x_2, \ldots, x_c > \in \mathscr{X}$. This work defines the hash function $h_d : \mathscr{X} \longrightarrow \mathscr{V}$ for the Bloom filters by

$$h_d(X) = d_1 \bullet x_1 \oplus d_2 \bullet x_2 \oplus \ldots \oplus d_c \bullet x_c \oplus d_{c+1}, \tag{3.5}$$

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 $\mathscr{X}'$ be a set of strings of $c+1$ bits. The hash function $h'_d : \mathscr{X}' \longrightarrow \mathscr{V}$, where

$$h'_d(X') = d_1 \bullet x'_1 \oplus d_2 \bullet x'_2 \oplus \ldots \oplus d_c \bullet x'_c \oplus d_{c+1} \bullet x'_{c+1}. \tag{3.6}$$

is a function in the H_3 class by definition, so the mapping to $\mathscr{V}$ is uniform. Since the key space of h_d, i.e., $\mathscr{X}$, can be viewed as a subset of $\mathscr{X}'$, 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 rate to $f = (1/2)^k$ [DKS04], the hardware complexity and simultaneous memory accesses also increase with a large k. Figure 3.4 presents the false-positive rate with respect to k and v/r. To 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. For restricted memory space, 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 Equation 3.3). We arbitrarily choose $b = 4$ herein because it fits well on a 32-bit

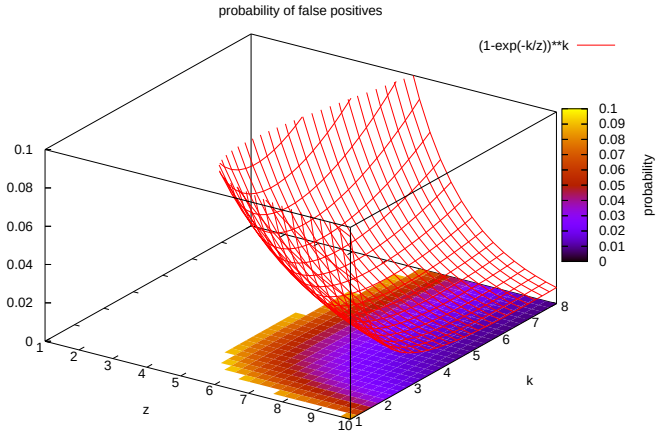


Figure 3.4: The false-positive rate with respect to k and the ratio of $z = v/r$.

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 $r = 30,000$.

3.3.6 The 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 $G_j, j = 0 \ldots m - b + 1$ 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

$$S_m = \begin{cases} \mathcal{P}(s=0)S_{m-1} + \sum_{i=1}^{m} \mathcal{P}(s=i)i & \text{for } m > b \\ \sum_{i=1}^{m} \mathcal{P}(s=i)i & \text{for } m = b \end{cases} . \tag{3.7}$$

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 $\mathscr{P}'$. The shift value is derived according to the position of the rightmost occurrence of B_0, so

$$\mathcal{P}(s=i) = (1-f)^i f \quad \text{for } i = 0 \ldots m-b. \tag{3.8}$$

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

$$\begin{aligned} \mathcal{P}(s=i) = \quad & (1-f)^{m-b+1}(1-\mathcal{P}(N_i)) \\ & \textstyle\prod_{j=m-b+1}^{i-1} \mathcal{P}(N_j), \\ & \text{for } i = m-b+1 \ldots m-1, \end{aligned} \tag{3.9}$$

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

$$\mathcal{P}(N_j) = 1 - (f + (1 - (1 - \frac{1}{|\Sigma|^{m-j}})^r)). \tag{3.10}$$

3. Neither B_0 is a factor of any pattern in $\mathscr{P}'$ nor its suffix is a prefix of any pattern. In this case,

$$\mathcal{P}(s=m) = 1 - \sum_{i=0}^{m-1} \mathcal{P}(s=i). \tag{3.11}$$

If the false-positive rate f is low enough, the probability in Equation 3.8 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.

3.4 Implementation details

3.4.1 Main components in the BFAST architecture

Figure 3.5 presents the three main components in the BFAST architecture. (1) The *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) The *verification interface* receives a verification job packed in a descriptor and fills an entry in the job queue. (3) The *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 $\mathscr{P}'$ 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

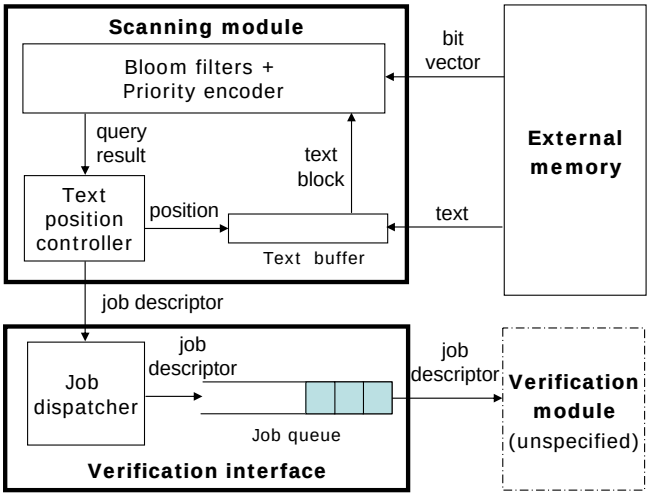


Figure 3.5: Overview of the modules in the BFAST architecture.

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 [Xil05]. Figure 3.6 illustrates the layout of the memory blocks to support multiple Bloom filters. Each memory block is configured as a 16k×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. Figure 3.6 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 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

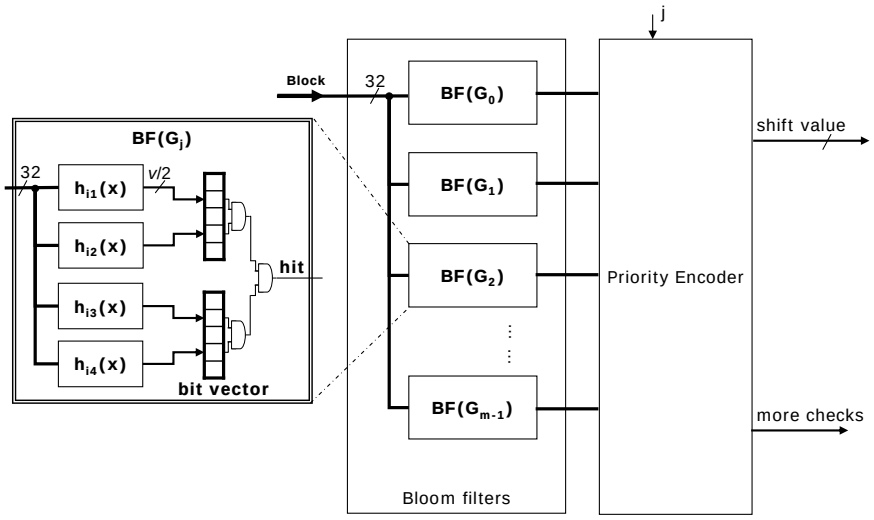


Figure 3.6: The layout of memory block to support multiple Bloom filters and the priority encoder.

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 3.3.4.

3.4.2 Pipelining design

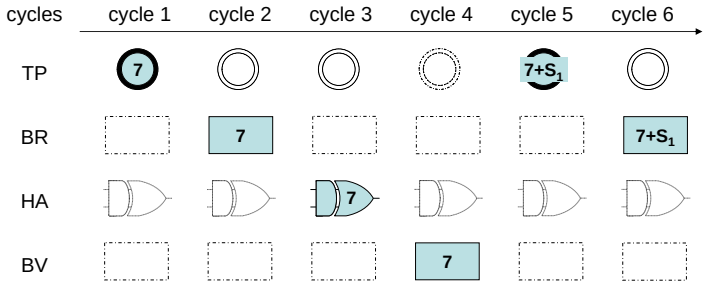
The process of deriving the shift value and moving the search window is divided into four phases for pipelining: text position controlling (TP), block reading (BR), computing hash functions (HA) and bit vector reading (BV). The text buffer is also logically divided into four segments. Assume the buffer length is ℓ, where ℓ is a multiple of 4. The four segments are located in the ranges of $[1, \frac{\ell}{4}]$, $[\frac{\ell}{4} - m + 2, \frac{\ell}{2}]$, $[\frac{\ell}{2} - m + 2, \frac{3\ell}{4}]$ and $[\frac{3\ell}{4} - m + 2, \ell]$. 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 $\mathscr{P}'$ must completely fall inside a segment. Figure 3.7 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.

3.5 Experimental results and comparisons

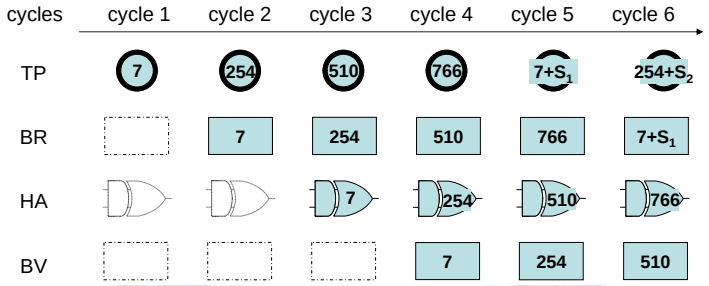
This work conducts a behavior simulation in C to estimate the performance, and runs a timing simulation in HDL to estimate the clock rate.

3.5.1 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



(a) Without pipelining.



(b) With pipelining.

Figure 3.7: The 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.

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 3.3.5. Both v and r are 2 to the power of some integer.

3.5.1.1 Performance for various number of patterns

Figure 3.8 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 $\mathscr{P}'$ happen to include the block '0x00 0x00 0x00 0x00', 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 1,000 patterns because the search window can advance only one character after a verification failure without the heuristic.

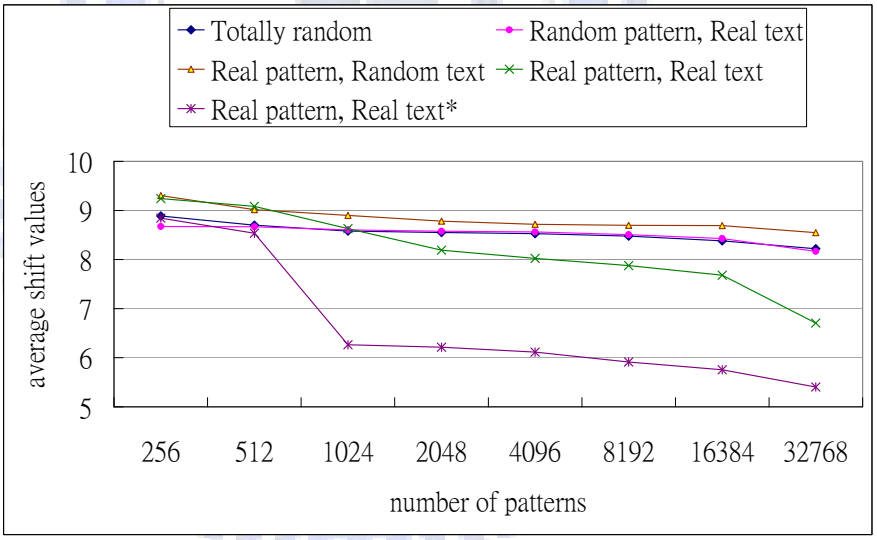


Figure 3.8: Average shift values for various number of patterns in four cases. An asterisk after the 'Real text' denotes the shift values are derived without the *bad-block* heuristic.

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.

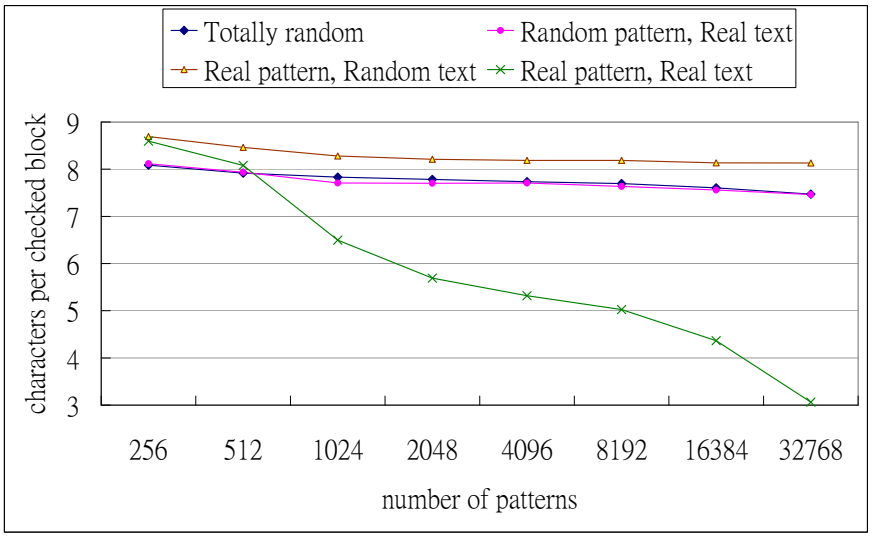


Figure 3.9: Average number of characters per checked block for various number of patterns in the four cases for $b = 4$.

The performance is estimated by s/n_s, which is the average number of characters per checked block. Figure 3.9 examines the values in the above cases. In the first three cases, the values are close to those in Figure 3.8, 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 $r = 8,192$, so effectively 5.03 characters are inspected in parallel.

We also test the average number of characters per checked block for various block sizes. Figure 3.10 presents the results. There are trade-offs in choosing the proper block size. A large block size has better performance, but may complicate matching a pattern shorter the block size. It depends on the actual need to choose the optimal block size.

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

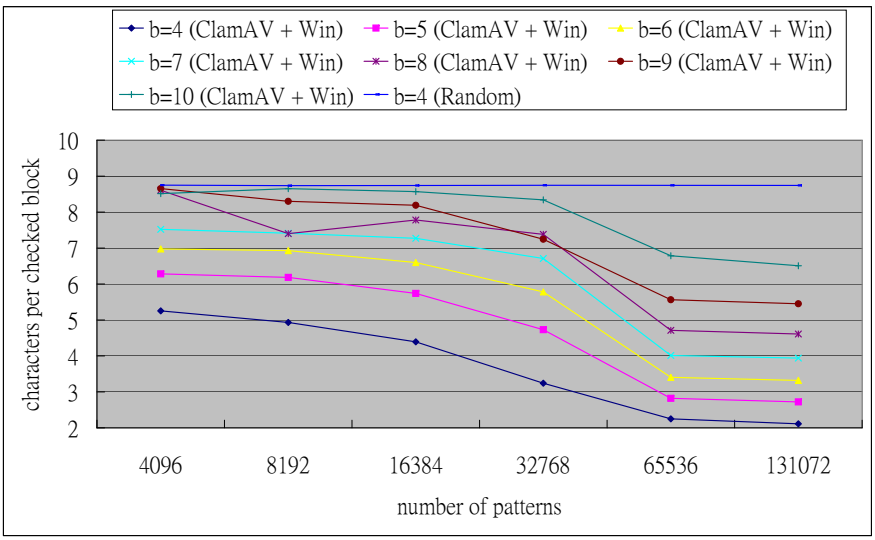
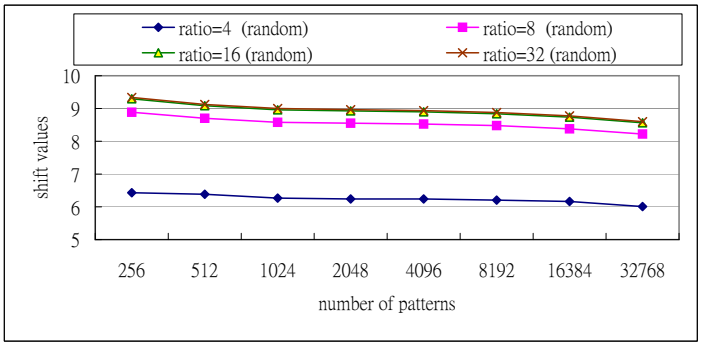


Figure 3.10: Average number of characters per checked block for various number of patterns in four cases for various block sizes.

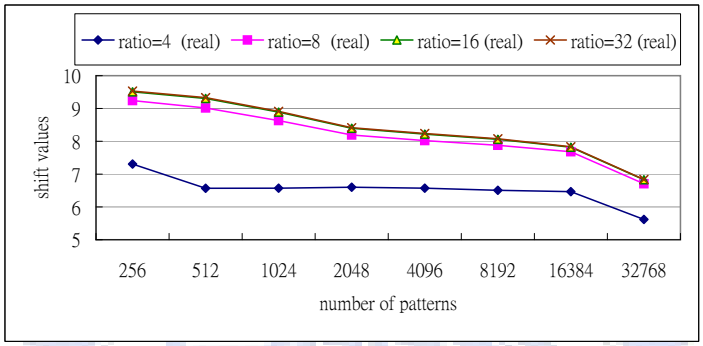
all the blocks in the search window are in their corresponding groups (e.g., the rightmost block in G_0) is tiny for $b = 4$. In the fourth case, Table 3.2 lists the average number of characters that have been scanned to meet a verification for various numbers of patterns. When $r = 8,192$ and $b = 8$, for example, the verification module has a margin of around 769 cycles (given 7.4 characters inspected in parallel effectively) to verify a match without blocking.

Table 3.2: The average number of scanned characters to meet a verification for various numbers of patterns in the practical case.

$r(b = 4)$	1,024	2,048	4,096	8,192	16,384	32,768
char.	1,156	1,111	578	561	187	55
$r(b = 8)$	4,096	8,192	16,384	32,768	65,536	131,072
char.	7,190	5,694	1,318	768	133	32



(a) random patterns and text.



(b) real patterns and text.

Figure 3.11: Average shift values for various lengths of bit vectors in the Bloom filters, where $v/r = 4, 8, 16$ and 32.

3.5.1.2 The impact of the length of the Bloom filters

Figure 3.11 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 3.3.5. 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

$$zr(m-2) + 65,536 + 256. \tag{3.12}$$

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 [Nor04]. They take 0.94 MB for 10,000 patterns, and had better be stored in the external memory.

3.5.2 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 4,096 patterns takes $2 * 8 = 16$ 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*4,096 = 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 [Xil05].

The system can operate at 150 MHz. The design utilizes 7,560 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 Figure 3.9), the throughput of the scanning module is up to 150*4.7*8 = 5.64 Gbps. If $b = 8$, an average of 7.78 characters can be effectively inspected in parallel for 16,384 patterns, and the throughput becomes up to 150*7.78*8 = 9.34 Gbps. If the signatures are properly specified, as discussed in Section 3.3.4, the worst-case throughput is 150*1*8 = 1.2 Gbps. It is suggested that long signatures of at least 15 characters be used in virus-scanning applications to avoid false positives [KA94]. In that case, the throughput could be higher because more characters are inspected per checked block for the long signatures.

3.5.3 Comparisons with other works

We categorize existing designs into filtering-based and automata-based architectures. Figure 3.12 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.

3.5.3.1 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 [DKS04] and [ADL04], inspecting G positions in parallel needs G sets of $L_{max} - L_{min}$ Bloom filters, where the pattern lengths range

architecture	BFAST	[DKS04]	[DL05]	[AC07]	[PP05]	[TS06]	[Lun06]
target application	anti-virus	IDS					
# of Bloom filters[a]	$m(=10)$	$G(t-m+1)$	k^2	N/A	N/A	N/A	N/A
# of text characters read in parallel	$b(=8)$	$G(t-m+1)$	k^2	k	variable	1	2
max. advance distance per iteration	m	G	k	1	2[e]	1[e]	2
# of patterns in the implementation[b]	16,384	35,475	2,259	1,655	≈1,500	1,000	≈2,000
embedded memory space	136 kB	400*4 kb[c]	754 kb	540 kb	288∼612 kb	0.4 MB	128 kB
external memory space	≈1.5 MB	unknown	a few MBs	unknown	0	0	0
platform	Xilinx Virtex II XC2VP30	Xilinx Virtex 2000E	Xilinx Virtex 4	Xilinx Virtex II Pro	Xilinx Virtex 2/4 , Spartan 3	unknown	Xilinx Virtex 4
number of parallel engines	1	4	8 (for $k=8$)	4 (for $k=4$)	2	# of split automata	2
operating frequency (MHz)	150	62.8	250	300	≈ 330	unknown	≈ 125
throughput (Gbps)	9.34[d](9.34)	2 (0.5)	14.1 (1.76)	10 (2.5)	1.7∼5.7 (2.85)	10 (1.25)	≈ 2 (1)

[a] Notations— G: number of parallel engines, t: maximum length (may be a threshold) of signatures, m: minimum length of signatures, k: number of positions 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.
[e] The value can be larger with higher hardware cost.

Figure 3.12: Comparisons between the BFAST and other architectures.

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 [SL05], but the number of Bloom filters is still $G(t-m+1)$ (e.g., 96 Bloom filters for $G=4$ in [SL05]). G is constrained by reading so many strings from the text simultaneously for parallel queries. The throughput is only around 2 Gbps 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] [DL05], 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

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

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 2,000 patterns are inside.

Papadopoulos and Pnevmatikatos [PP05] 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. [SPW05] 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 TriBiCa (Trie Bitmap Content Analyzer) that builds a trie structure to identify whether a window of characters belong to a set or not [AC07]. 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 Gps at 300 MHz. Four engines should work in parallel can achieve 10-Gbps throughput. The Snort signature set in this design takes around 540 kb of memory.

3.5.3.2 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 [TS06] 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. [SIH04] 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. [TLLar, LTLar, LTH07, TLL05] build root-index tables to derive the next state after several characters from the root state, and pre-hashing 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 [Lun06] 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.

3.5.3.3 Linear time vs. sub-linear time

We do not intend to compare with all of existing architectures, 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 anti-virus applications typically have long patterns [KA94], 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 3.3.4, 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 $2,000 \sim 3,000$ 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.

3.6 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 Gbps for more than 10,000 virus signatures, while the worst throughput is 1.2 Gbps with properly specified signatures. If the block size is eight characters, the throughput can be up to 9.34 Gbps for 16,384 patterns. 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 light-weight 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.

CHAPTER 4

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

4.1 Introduction

Scanning the content on network or storage devices for viruses involves computationally intensive string matching against a pattern set of virus signatures. Although designing an efficient method for high-speed content inspection has sparked a number of innovations lately, most of them look to hardware approaches that offload string matching to a specialized hardware engine [LLLar], especially for Snort-style intrusion detection (`www.snort.org`); however, as many anti-virus applications run on software environment (e.g., a commodity computer), deploying a hardware accelerator is costly and inflexible. Compared with intrusion detection, anti-virus applications are 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 anti-virus scanning.

Modern computer architecture brings new challenges to software implementation. A compact data structure to improve cache locality becomes critical because of the "memory wall" — memory access is slow [WM95]. Anti-virus applications have a much larger pattern set than Snort, which has only thousands of patterns. For example, ClamAV (`www.clamav.net`) has claimed a set of more than 200,000

patterns. A large pattern set not only demands large memory space, but also significantly slows down string matching. Carefully tuning the data structure is getting critical to the performance.

A common class of methods track a finite automaton that accepts the patterns in the pattern set, such as the Aho-Corasick (AC) algorithm [AC75]. The tracking generally reads one character in the text per iteration. Although some can track multiple characters per iteration with hardware assistance for high performance [DL05,SIH04], implementing them in software is not so efficient. The data structure of the automaton contains the transitions from each state and the failure links, and should be compressed in a compact representation [TSC04,Nor04]. 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 pattern set in anti-virus applications is much larger than that in intrusion detection, and virus signatures are generally long (may be up to hundreds of characters) to avoid false positives, making compression even challenging.

Another class of methods moves a search window through the text to check whether it contains a suspicious match or not [EC05, WM94]. Assuming most of the text is legitimate, these methods can quickly exclude the legitimate text, and verify only the suspicious matches. The patterns can be represented in a compact data structure such as a shift table or a Bloom filter. There is a tradeoff in deciding the window size [EC05]. A large window size is preferred because matching a long window implies large likelihood of a true match and thus reduces the verification frequency. However, matching a short pattern within the window becomes difficult. Some of the methods 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 algorithm [WM94]. They are generally fast, but have the Achilles' heel — the maximum skip distance is bounded by the shortest pattern length in the pattern set. These methods therefore have the problem with short patterns.

According to the above observation, either class of methods has its limitations. Because most of the patterns in anti-virus applications are long to reduce false positives [KA94], we can exploit the feature to increase the efficiency while reducing the memory requirement. 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 can scan for only long patterns to derive long shift distance of the search window. The character distribution in both the patterns and the text is non-uniform, making the shift distance shorter and the verification frequency higher than those in theoretical analysis, so the performance is slowed down. The backward-hashing mechanism can effectively reduce the verification frequency and exploit long shift distance if there is a chance. After the partition, the AC algorithm can scan only the relatively small set of short patterns. The data structure of the automaton is compact and saves the memory space. The method is applied to ClamAV to improve its performance. Some factors in software implementation such as cache locality will drastically affect the overall performance, and this work will also discuss them in practical implementation.

The rest of this paper is organized as follows. Section 4.2 reviews the existing work for string matching for virus scanning. Section 4.3 presents the details of the hybrid method and the practical implementation issues, followed by the performance evaluation of the algorithm in Section 4.4.1. Section 4.5 concludes

this work and points out future work.

4.2 Review of existing work

4.2.1 String matching algorithms

Scanning the text for multiple patterns typically tracks the partially matched prefixes with a finite automaton that accepts the patterns, or filters the text with a search window to weed out unsuccessful matches and verifies only suspicious matches. The former features the deterministic execution time that guarantees the worst performance even though algorithmic attacks are present to exploit the worst case of an algorithm, but may require large memory space to store the transition information. The latter features memory economy and fast average execution time, but must carefully deal with possible attacks.

Recent automaton approaches focus on fast tracking a compressed automaton with hardware assistance [DL05, Lun06, AC07, TS06, BCT06], but their software implementation is not as efficient as the hardware counterpart. Although some compression methods are independent of hardware [Nor04, YCD06], their scalability to a large pattern set of long virus signatures could be a problem. First, the transition table is not so sparse due to the large pattern set. Second, the method to simplify repetitions in the patterns [YCD06] is unable to compress the characters in the long patterns.

A filtering approach can map the search window along the text with one or more hash functions to see whether the window matches an entire pattern or part of a long pattern. If a suspicious match occurs, whether a true match with that pattern is verified. This approach is very memory efficient because a pattern is stored as only a hash value or the bits in a few addresses. The 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 [EC05]. Moreover, the window can slide by only one character per iteration.

The algorithms such as the WM algorithm can map only the suffix of the search window, ignore the remaining characters in the window, and shift the window to the next position according to certain heuristics in the search stage. To be specific, suppose the window is m characters long, and let X be the block of b characters in the window suffix. The heuristic can look up the block in a shift table (built in the preprocessing stage) 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 in 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.

The maximum shift distance is $m-b+1$, meaning it is bounded by the shortest pattern length. Although using a heuristic like that in [**?**] allows the maximum distance up to m characters, building the shift table becomes extremely time-consuming for large b because up to $|\Sigma|^b$ possible blocks should be considered to build the table, where Σ is usually the ASCII character set and $|\Sigma|$ is 256.

Mapping multiple blocks to the same entry, in which the minimum shift value of these blocks is filled in, can compress the shift table [WM94]. There are tradeoffs in the table compression. Reducing the table size can improve the cache

locality, but at the cost of smaller shift values (due to the minimum shift values of multiple blocks) and more frequent verification (due to the increasing likelihood of a zero shift value). On the contrary, expanding the shift table can derive larger shift values, but at the cost of worse cache locality. Therefore, the table size should be carefully tuned to achieve the optimal performance. Moreover, the non-uniform character distribution in the text and the patterns means that some characters or blocks appear more frequently than that in probabilistic analysis. A frequent block may happen to be the suffix of some pattern, resulting in frequent verification. The possibility increases with the size of pattern set. This problem should be solved, or the verification frequency may be high.

4.2.2 Virus signatures and string matching in ClamAV

The virus database in ClamAV grows very fast lately, containing more than 200,000 signatures at the time of writing (April 2008). The database contains four types of virus signatures: basic patterns, multi-part 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 multi-part pattern consists of multiple parts of basic patterns to be matched in sequence for virus identification. The former is sufficient to detect non-polymorphic 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. MD5 matching spends most of the time in MD5 computation, and then checks whether the 16-byte output is a MD5 signature. Phishing matching checks whether a URL is in a URL list. The latter two types are beyond the scope of this work because matching them differs from ordinary multiple string matching that scans long text for multiple patterns.

Table 4.1: The number of parts or basic patterns and their minimum/maxmum lengths in each target type.

Type	Aho-Corasick (AC)			Wu-Manber (WM)		
	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
Subtotal	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
Total	12,796	2	176	224,946	4	392

The patterns can come with context information such as target file type, virus position in the text and so on to reduce false positives. For example, the signature `W32.Deadc0de` is a four-character basic pattern: `0xdec0adde`. The pattern is required to start from the 64th byte in a file of Portable Executable (PE) format, or the match will not be claimed. ClamAV separates the patterns other than generic ones into individual data structures according to the target type. Therefore, after determining the target type of the text, ClamAV can scan for only the patterns associated with that type, besides the generic patterns. The current version of ClamAV (version 0.92) scans the text with both the AC algorithm for parts in the multi-part patterns and the WM algorithm [1] for basic patterns. Table 4.1 summarizes the number of parts or basic patterns, as well as their minimum/maximum lengths for the two algorithms in each target type.

An old version of ClamAV simplified the AC automaton to a trie structure up to a maximum height h. The patterns with identical prefixes of h characters are

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

stored in a linked list pointed by a leaf node at level h. Because the minimum pattern length for the AC algorithm is only two characters, h was set to 2, and the linked list was increasingly longer as the pattern set grows. Traversing the long linked list is time-consuming. Miretskiy et al. [MDW04] proposed a trie structure that can stores a pattern at the lowest possible level as soon as the pattern's unique prefix is identified, but the automaton still needs the space to accommodate thousands of patterns. The scalability issue still remains unsolved. 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 [EC05] presented a filtering approach named Hash-AV to weed out most of the legitimate text with a Bloom filter [Blo70], which can reside in the L2 cache due to its space efficiency. The design selects a window of seven characters for the filtering and four hash functions to reduce the number of false positives. Because the Bloom filter is unable to handle the patterns shorter than seven characters, they are left to the AC algorithm for multi-part 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. Using only three characters to derive the distance also results in high false-positive rate. However, we believe the skipping is still beneficial to high performance if the search window could be somehow lengthened. Moreover, Hash-AV does not improve the AC algorithm in ClamAV at all.

4.3 The hybrid algorithm and practical issues

This work partitions the patterns in ClamAV into long and short ones. The BH algorithm is responsible for only long patterns to lengthen the average skip distance, while the AC algorithm scans for only short patterns to reduce the automaton sizes.

4.3.1 The BH algorithm

The WM algorithm implementation in ClamAV faces several practical performance issues. First, the block to derive the shift distance consists of only three characters (i.e., $b = 3$). Although the block size seems sufficient to weed out most false positives (i.e., suspicious matches that should be verified) from a probabilistic aspect, it is not the case in practice due to non-uniform block distribution. For example, the block '0x00 0x00 0x00' is frequent in Windows executable files. The false-positive rate will be higher than expected if the block is the suffix of some pattern. For example, the rate is around 37% in our study on a sample set of Windows executable files. Extending the block size can reduce the false-positive rate, but it 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 [?] 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 pattern set has a pattern as short as only four characters (See Table 4.1). Skipping longer than the shortest pattern length may miss the shortest pattern because it may happen to appear in a position between two consecutive skips.

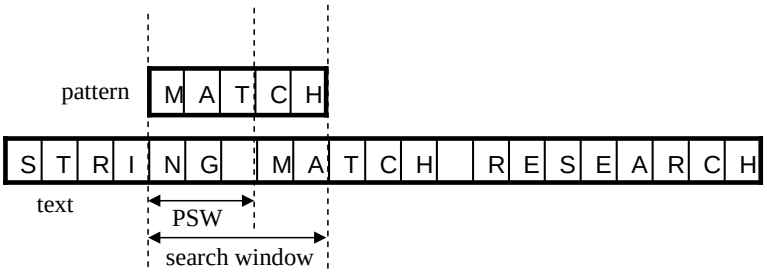


Figure 4.1: The illustration of a missed match.

This factor restricts the effectiveness of applying the WM algorithm to ClamAV.

We use the following methods in the backward hashing (BH) algorithm to solve the aforementioned problems.

4.3.1.1 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 does not appear in the pattern set, 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) [LHC04], but the forward search may result in false negatives. Figure 4.1 illustrates an example to show the shift should not go beyond the PSW. If the characters beyond the PSW are not examined, no match should be excluded. In this example, the pattern 'MATCH' will be missed if the shift distance is longer than three characters. The BH algorithm looks backward in the search window instead. Because the characters in the suffix have been examined, shifting beyond them is safe.

The new heuristic is formally stated as follows.

1. If neither X appears in the pattern set nor any suffix of X is a prefix of any pattern, the shift value can be m if $m \geq b$, or b otherwise.

2. X does not appear in the pattern set, 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 $m - k$ if $m \geq b$, or $b - k$ otherwise.

3. X is a substring of some pattern if $m \geq b$, or some pattern is a substring of X otherwise. In the former case, the shift value is $m - j$, assuming the rightmost occurrence of X ends in position j of some pattern. If $j = m$, X is the suffix of some pattern, and whether a true match occurs should be verified, after which the search window is shifted by one character. In the latter case, a match is claimed.

The shift value for every different X is calculated and stored in a shift table in preprocessing, so a simple table lookup can derive the shift distance, just as simple as that in the WM algorithm. The maximum shift value is m, rather than $m - b + 1$.

Like the WM algorithm, the BH algorithm builds a shift table in the preprocessing stage according to the above heuristic. Suppose a block X is mapped to the table with the hash function h. The steps 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 $1 \leq q < \min(m, b)$, 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.

4.3.1.2 The bad-character heuristic

Increasing the block size b can reduce the false-positive rate, but the increase also complicates the build-up of the shift table. For example, the preprocessing stage needs to map all possible blocks whose last character (a suffix) is also the first character (a prefix) of some pattern to fill in the shift value of $m - 1$. For each pattern, totally $|\Sigma|^{b-1}$ blocks have the last character that is also the first character of that pattern. In other words, the number of blocks increases exponentially to b, making building the shift table with a large block size very expensive in preprocessing.

The bad-character heuristic intends to reduce the false-positive rate while keeping the block size manageable. Moreover, it attempts to exploit a large shift value if possible. The block size is still 3, while the heuristic avoids immediate verification by checking the additional blocks B_1,B_2,...,$B_{\lfloor m/b \rfloor}$ to exploit a larger shift value if needed, where B_j denotes the j-th non-overlapping block counted backward from the window suffix. We give a trivial example in Figure 4.2 to justify the derivation of a safe shift from B_j. Suppose the algorithm scans for only a pattern `XAMPAMPLE`. The rightmost block `PLE` in the search window matches the the pattern suffix, so the hashing goes on to the next block `XAM`, whose position is 3 characters away from its rightmost appearance in the pattern. Therefore, shifting the search window by 3 character is safe.

The heuristic also looks up B_j in the shift table. Let SHIFT[B_j] be the shift value derived by mapping B_j to the table. Because the shift values in the table are computed with respect to B_0, they are not directly applicable to B_j. However, we still have a chance to exploit the shift values for B_j. In the above example,

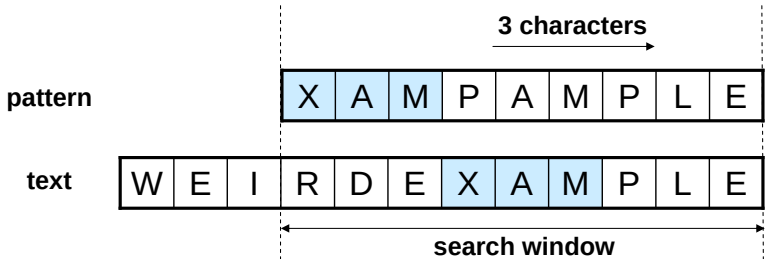


Figure 4.2: The heuristic in the bad-block heuristic.

SHIFT[XAM] is 6 because XAM ends at position 3 and the pattern length is 9. From the value in the shift table, we can derive XAM's position and infer the safe shift distance.

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 is filled in. Therefore, 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 implies only the position of a block's rightmost occurrence, and it losses the exact information such as whether a block appears in a specific position or appears multiple times in the patterns. Figure 4.3 illustrates an example to illustrate the information lost in the shift table. In this example, because PLE is the pattern suffix, SHIFT[PLE] is 0. When the heuristic checks the block B_1 =PLE in the table, it knows only that PLE appears in the pattern suffix. Whether PLE also appears at position 4–6 is unknown from the table, 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 has better be conservative and keeps on checking B_{j+1} for not missing any match. 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.*

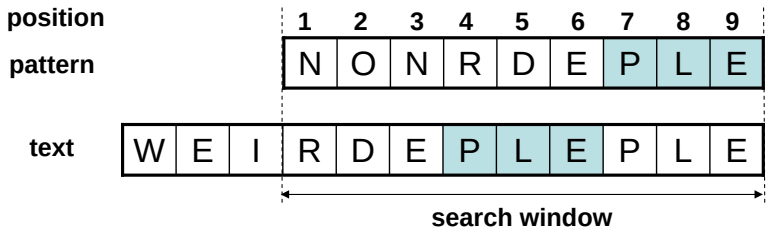


Figure 4.3: An example to illustrate the information lost in the shift table.

Proof. Suppose a match occurs when the search window is shifted by a shift s, where $s < \text{SHIFT}[B_j] - jb$. This means there is a B_j that ends at position $m - jb - s$, which implies

$$\begin{aligned} \text{SHIFT}[B_j] &\leq jb + s \\ &< jb + \text{SHIFT}[B_j] - jb \\ &= \text{SHIFT}[B_j]. \end{aligned} \tag{4.1}$$

Equation 4.1 leads to a contradiction, i.e., if the search window is shifted by less than $\text{SHIFT}[B_j]-jb$, no match should occur. A shift of $\text{SHIFT}[B_j]-jb$ is thus safe. □

4.3.2 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 whose lengths are longer than or equal to ℓ in the AC algorithms 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 only difference is that the AC algorithms need to track multiple parts of basic patterns in the multi-part patterns after the individual parts are matched. In the hybrid method, therefore, the patterns transferred to the AC algorithm can be thought of as single-part patterns with

Table 4.2: The number of parts or basic patterns in each target type after sorting out them.

Type	$\ell = 9$		$\ell = 12$		$\ell = 15$	
	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

optional context information, and the multi-part tracking function is duplicated in the BH algorithm when the individual parts are found.

Because the block size is 3, the length threshold is chosen to be a multiple of 3 for easy implementation. Here we set $\ell = 9, 12$ and 15, and sort out the patterns according to the threshold. Table 4.2 lists the number of parts or basic patterns in each target type after the re-arrangement. In the table, the number of AC patterns is only 24.7% $\sim$ 41.4% of the original number on average, meaning the pattern set becomes only one-fourth to half of the original set. Although the decreased patterns are moved to the BH algorithm, they contribute just 9.1% $\sim$ 11.7% more patterns to it. Given that the maximum shift distance is three to five times longer, the change is still beneficial.

4.4 Parameter selection and evaluation

4.4.1 Parameter selection

The hybrid method should properly chooses 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 subsections 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 2x4 MB L2 cache on the chip. The characteristics of the patterns from ClamAV version 0.92 has been list in Table 4.1. We collected a set of Windows executable files of around 70 MB to be scanned for the patterns because the generic signatures and those for files of Microsoft Portable Executable (PE) format dominate the total pattern set in ClamAV.

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 characters in a block. The search window consists of only three characters due to the shortest pattern length[2]. According to the heuristic in the WM algorithm, the shift values are very short, consisting of only 0 and 1. Around 46.6% of the entries are 0 in the shift table for generic signatures, and around 75.9% are 0 for signatures associated with MS PE format. Therefore, the average shift value is only 0.28 characters on average, meaning that most of the scanning time is spent in verification and the benefits of skipping in a typical WM algorithm is sacrificed.

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 $h(x_1, x_2, x_3) = 422x_1 + 74x_2 + x_3$, quadru-

[2]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 4.3: 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

pled 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 the cache locality. The length thresholds are set at $\ell = 9, 12$ and 15. Table 4.3 presents the shift values for various cases in the experiment, where 1x denotes the original table size in ClamAV implementation, 2x denotes double the size, and so on.

Two observations are in Table 4.3. 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 verification frequency is significantly decreased. The backward hashing checks all the characters in a long search window, and can reduce the probability of verification, which is invoked only when checking every blocks in the search windows derives a shift value of 0.

Table 4.4 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 4.3, the differences in the execution time is insignificant. Because of the non-uniform character distribution, checking only the rightmost block, as that in the WM algorithm, could lead to frequent verification, but the BH algorithm checks every block in the search window. The insignificance of the execution

Table 4.4: 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

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, and 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 shows $\ell = 9$ is a proper length threshold. The AC algorithm is still much slower than the WM algorithm, and a bottleneck to be further improved in the future work.

4.4.2 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.

Because the BH algorithm, which reduces the verification frequency, is much faster than the ClamAV implementation, even when the table size is larger, which implies worse locality. For example, when the table size is eight times larger in Table 4.4, the BH algorithm is still much faster than the original implementation. We can infer 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 both Table 4.3 and Table 4.4, 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 benefits 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 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 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 long search window.

Table 4.5 compares the throughput between the original method and the hybrid method. The BH algorithm is 109 times faster than the original WM algorithm, due to the longer shift distance and the verification frequency. Despite the impressive acceleration for scanning non-polymorphic viruses, the improvement

Table 4.5: Comparing the throughput (Mb/s) 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
Only AC	37.19	55.44
WM/BH+AC	19.56	54.72

over the AC algorithm is only 49% faster for polymorphic viruses. Because virus scanning contains two passes of the WM/BH algorithm and the AC algorithm, the overall improvement of the hybrid algorithm is 2.8 times faster.

4.4.3 Discussion of worst-case performance

It is theoretically possible to dramatically reduce the performance of a sub-linear time algorithm such as the BH algorithm in some extreme cases. For example, if the pattern set has a pattern `aaaaa` and the text consists of all `a`'s, then a verification is needed for every shift of only one character in the text, and the time complexity becomes super-linear. Although worst performance in linear time for such algorithms is possible for single-string matching [Gal79], it is non-trivial 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 algorithmic attack is likely to happen, and an alarm is raised. By carefully scheduling the process of a slow virus scanner, the negative impact from a slow process can be contained [EC05].

4.5 Conclusion

This work presents a hybrid algorithm that combines the backward hashing (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 is an efficient one to combine the benefits of the traditional AC algorithm and the WM algorithm.

As multi-core processors become popular in the market, how to exploit the parallelism in the processors to accelerate virus scanning becomes interesting. Because the data structure is independent in both algorithms, and the pattern set in ClamAV is classified according to the target type, virus scanning could be a good candidate for parallelization on a multi-core processor. When the files or network traffic consists of a mix of content that belongs to various types, there is a potential of scanning each type on different processor cores. The two algorithms can also independently run on two separate processors. The method to explore the potential of parallelism is left for further study.

CHAPTER 5

Accelerating Web Content Filtering by the Early Decision Algorithm

5.1 Introduction

A huge amount of Web content is widely accessible nowadays. As inappropriate content such as pornography proliferates with the growth of World Wide Web, access control of the content is demanded in some situations. For example, an employer does not want the employees to watch stock information during working hours, or parents do not want their children to browse pornographic content. Web filtering products that enforce access control are therefore popular on the market. They can be deployed either on a host computer (e.g., in a family), or on the gateway for central management in a company or an Internet service provider.

Four major approaches are generally adopted in Web filtering nowadays: Platform for Internet Content Selection (PICS), URL-based, keyword-based and content analysis [LHF02]. According to a recent review of up-to-date Internet filters, commercial products have widely adopted content analysis besides the URL-based approach (`internet-filter-review.toptenreviews.com`). Content analysis automatically classifies the Web content into a category first, and then makes the filtering decision, either to block or to pass the content, according to the management policy. The analysis generally complements the URL-based

approach to relieve the effort of frequently updating the URL list and to reduce the number of false negatives due to an outdated URL database.

The efficiency of content analysis algorithms is essential due to their complexity. Slow analysis in Web filtering leads to long user response time and also degrades the throughput of Web filtering systems. We therefore focus on *text classification*, which remains an important and efficient approach to Web content analysis, despite the research on image content analysis for Web filtering [Wan01, Wan]. Moreover, image content analysis in Web filtering is mostly designed for pornography recognition, but not as effective for content in other banned categories.

Numerous text classification algorithms with high accuracy have been proposed. They are designed primarily for off-line applications, such as Web categorization for catalogs hosted by Internet portals. The research on these algorithms mostly emphasizes on classification accuracy, but their efficiency on execution is rarely addressed. However, their efficiency should deserve attention for on-line applications such as Web filtering so that text classification will not slow down these applications significantly.

This work presents a simple, but effective *early decision* algorithm to accelerate Web filtering. The algorithm is based on the observation that it is possible to make the filtering decision before scanning the entire content, as soon as the content can be confirmed with a high probability that it really belongs to a certain category. The fast decision is particularly important, since most Web content is normally allowed and should pass the filter as soon as possible.

5.2 Related Work in Web Filtering

5.2.1 Approaches of Web Filtering

A Web filtering system can either block HTTP requests according to their URLs, or block the Web content using several approaches to be discussed later. The former approach maintains a large database of banned URLs. If the URL in a request is found in the database, the request is blocked. The database is frequently updated by the collaborative effort of human reviewers.

URL blocking is very efficient in processing, and the content on the banned sites will not occupy the bandwidth of the download link. However, since Web sites on the Internet change very often and new sites grow extremely fast, the database is unlikely to keep pace with the dynamic change of Web sites. Hence the system may fail to ban some sites that should be banned.

Blocking the Web content can remedy the insufficiency of URL blocking. Several types of information can help to determine whether a Web page should be blocked. The Platform for Internet Content Selection (PICS) specification (`www.w3.org/PICS/`) allows the content publishers to rate and label Web content so that a Web filtering system can identify the category and judge the offensiveness of the content according to the PICS information. However, labeling the Web content is voluntary. Publishers of banned content may not want to label the content and let their sites be banned. Hence a system cannot rely solely on the PICS information to judge whether the content should be blocked or not.

Another simple approach is looking for offensive keywords in the Web content. The keywords should be carefully selected, or false positives are likely to happen. For example, using 'sex' as a keyword could possibly block Web content about sex education. Therefore, detailed content analysis is often desired rather than

simple keyword blocking.

Content analysis is generally based on machine-learning methods. It involves looking for representative features that tell the category of the content. The features could be keywords, hyperlinks, images and so on [HCC06]. The classifier on a Web filtering system first learns the features from a training set of Web pages collected from both banned and allowed categories off-line. After the learning, the classifier is able to judge the Web content according to the features. Most features are text-based in practice, because text classification is more efficient than image analysis and can classify content in categories other than pornography.

5.2.2 Text Classification Algorithms

Text classification is an important part in Web filtering because the text in Web content provide rich features for filtering. Yang et al. and Sebastiani [YL99,Seb02] surveyed and compared comprehensively existing text classification algorithms, such as support vector machine (SVM), k-nearest neighbor (kNN), neural network (NNet), decision tree and naïve Bayesian (NB). These algorithms are shown to achieve around 80% or higher in accuracy, measured by the harmonic average of recall and precision, where recall is defined to be the ratio of the number of correct positive predictions divided by the number of positive examples, and precision is the ratio of the number of correct positive predictions divided by the number of positive predictions.

Support vector machine (SVM) uses a process to find a decision surface that can separate positive examples and negative examples in a multi-dimensional feature space, in which training documents are represented as vectors. SVM is efficient and can handle a large number of training examples, but it would be difficult to find a kernel function to map vectors to the multi-dimensional space so

that both positive and negative examples are roughly linearly separable [HCC06].

The k-nearest neighbor (kNN) method labels the training examples in the feature space to their categories. In the classification stage, the kNN method selects k most similar documents (measured by the distance in the feature space) in the training examples, and assigns the test document to the label that is the most frequent one among the k nearest training documents. The kNN method is very simple, but it had better allow the possibility that a document is assigned to more than one category, or the accuracy may be degraded [YL99].

The neural network method (NNet) has been widely studied in artificial intelligence. A neural network is an adaptive system that consists of a group of interconnected artificial neurons. The system can be trained to change its internal states to reflect the association of the documents and their categories. Although neural networks are efficient in handling both linearly and non-linearly separable examples, the classification decision is not easily understandable because of the "black-box" nature of the neural network.

A decision tree represents a test on an attribute as an internal node and the test results as outgoing branches of that node. A document is classified by traversing from the root node to a set of internal nodes successively based on the test results on the attributes, until a leaf node is reached. The decision is easily interpretable, but requires to carefully choose the attributes to avoid the over-fitting problem.

Naïve Bayesian classification uses a probabilistic model, in which the probabilities of words in a document that belongs to each category are estimated. The probabilities can be used to estimate the most likely category of a test document. We leave the details of NB in Section 5.3.2 because our work is based on NB.

Some research work has attempted to exploit the structural information, such

as hyperlinks and meta-information to improve the accuracy [GTL02, AGS99]. These methods require parsing the Web content to extract the semantical information. This direction is currently beyond the scope of this work because the parsing takes more time than our model of viewing the text as a sequence of words, and may not be so efficient for real-time filtering.

5.3 The Early Decision Algorithm

This key idea of the *early decision* algorithm to accelerate Web filtering is that making the filtering decision is possible before scanning the entire content, as soon as the content is confirmed to really belongs to a certain category with a high probability. The fast decision is particularly important for on-line filtering because most Web content is allowed and should pass the filter as soon as possible.

Among the aforementioned algorithms in Section 5.2.2, we herein choose naïve Bayesian classification to be the basis of the early decision algorithm because its computation can be easily turned into score accumulation. The probability that a test document belongs to each category can be easily estimated as the classifier scans along the document. However, we believe that NB classification is by no means the only algorithm that can make an early decision from part of the Web content. Other text classification algorithms can follow the similar principle to be introduced in this section to accelerate Web content classification.

5.3.1 Keyword Distribution

The early decision algorithm can make the filtering decision as soon as possible, by scanning only the front part of the Web content. Because the algorithm does not have to scan the entire content, the filtering is much faster. The trick is just

like one may not have to wait until the end of a speech to know its topic, as long as the front part of the speech provides sufficient information about the topic.

The feasibility of the early decision algorithm comes from the observation that the front part of the Web content has adequate keywords to indicate whether the content should be blocked or passed. The keywords herein are defined to be words from the banned categories with high "information gain" (`www-2.cs.cmu.edu/~mccallum/bow/rainbow`), which, simply put, measures how indicative a word is to help distinguish the category of content from the others. To justify the feasibility, we investigated the keyword distribution in typical Web content collected from the YAHOO directory service (`www.yahoo.com`). Figure 5.1 presents the average distribution in Web content of both the banned and allowed categories. On the horizontal axis is the keyword position normalized by the content length of each Web page. The position is represented in percentage. For example, if a keyword is the 50-th word in a 100-word Web page, then its position is at 50%. On the vertical axis is the appearance probability of keywords in the positions throughout the Web content. The probability is also represented in percentage.

According to Figure 5.1, the keywords in the banned categories start to appear more frequently than those in the allowed categories since the front part of the Web content. In other words, keywords from the front partial content can provide the clue to identify the category that the entire content belong to. Therefore, it is feasible to make the filtering decision before scanning the entire content.

Although the above observation is generally true in normal conditions, a malicious user may deliberately stuff irrelevant content in the front part of the Web page to deceive the filter. The deception is not difficult to avoid. First, the early decision algorithm strips the HTML tags during the content analysis because the tags are generally used to specify the attributes of the content, and

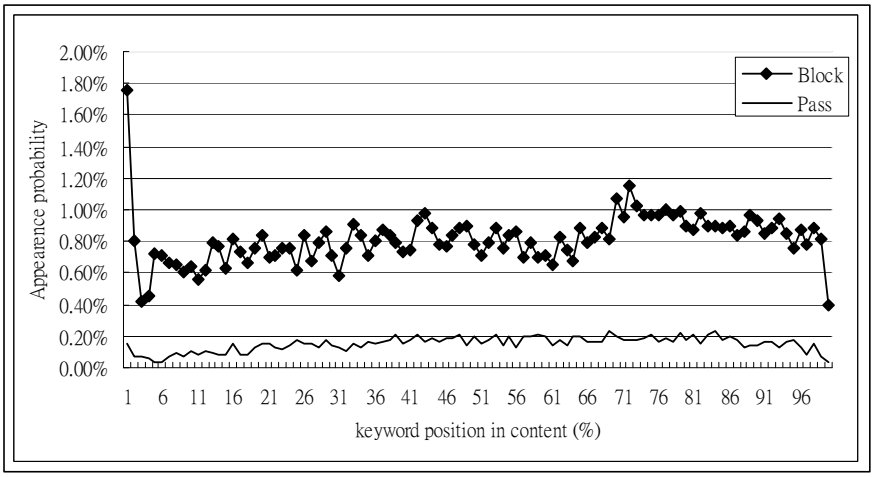


Figure 5.1: The keyword distribution in the Web content of both the banned and the allowed categories.

will not be displayed on the browser. If the irrelevant content is hidden inside the HTML tags, it will be ignored and unable to deceive the filter. Second, if the irrelevant content is in the Web text outside the tags, it will be displayed on the browser, and will also confuse the viewer who browses the content. This deception approach will lead to a great limitation on the layout design of the Web pages.

5.3.2 Naïve Bayesian Classification

The NB classification is divided into two stages: training and classification. In the training stage, the classifier learns the probabilistic parameters of the generative model from a set of training documents, $D = \{d_1, \ldots, d_{|D|}\}$. Each document consists of an ordered sequence of words from a vocabulary set $V = \{w_1, w_2, \ldots w_{|V|}\}$ and is associated with some category from a set of categories $C = \{c_1, c_2, \ldots, c_{|C|}\}$. Two types of parameters are included in the model [Mit96]: (1) $P(w_t|c_j)$: the estimated probability that word w_t appears in the documents of category c_j and

(2) $P(c_j)$: the estimated probability of category c_j in the training documents. The former parameter is derived by

$$P(w_t|c_j) = \frac{1 + \sum_{i=1}^{|D|} N_1(w_t, d_i|d_i \in c_j)}{|V| + \sum_{t=1}^{|V|} \sum_{i=1}^{|D|} N_1(w_t, d_i|d_i \in c_j)}, \tag{5.1}$$

where $N_1(w_t, d_i)$ is the times word w_t appears in document d_i, and

$$P(c_j) = \frac{1 + \sum_{i=1}^{|D|} N_2(d_i, c_j)}{|C| + |D|}, \tag{5.2}$$

where $N_2(d_i, c_j)$ is 1 if $d_i \in c_j$, or 0 otherwise.

Notably, the above two equations are filtered by Laplace smoothing to avoid estimating probabilities to be zero. In the classification stage, the posterior probability $P(c_j|d_i)$ that a test document d_i belongs to each category c_j is computed. The category c_j that maximizes $P(c_j|d_i)$ is the one that document d_i belongs to most likely. $P(c_j|d_i)$ is derived by

$$\begin{aligned} P(c_j|d_i) &= \frac{P(c_j)P(d_i|c_j)}{P(d_i)} \\ &= \frac{P(c_j)\prod_{k=1}^{|d_i|} P(w_{d_i,k}|c_j)}{P(d_i)}, \end{aligned} \tag{5.3}$$

where $w_{d_i,k}$ is the k-th word in document d_i. The document d_i is viewed as an ordered sequence $< w_{d_i,1}, w_{d_i,2}, \ldots, w_{d_i,|d_i|} >$, with the assumption that the probability of a word occurrence is independent of its position in the document, given the category c_j. Therefore, $P(d_i|c_j)$ can be written as the product of $P(w_{d_i,k}|c_j)$, for $k = 1 \ldots |d_i|$. Taking the logarithm on both sides of Equation 5.3 simplifies the computation of the posterior probability $P(c_j|d_i)$ from a series of multiplications to a series of additions. The computation then becomes

$$logP(c_j|d_i) = log\frac{P(c_j)}{P(d_i)} + \sum_{k=1}^{|d_i|} logP(w_{d_i,k}|c_j). \tag{5.4}$$

Since the logarithm function is increasing and $log\frac{P(c_j)}{P(d_i)}$ is kept constant in the classification stage, accumulating only the term $logP(w_{d_i,k}|c_j)$ during text

scanning is sufficient to derive the maximum of $P(c_j|d_i)$. The score of each word $w_t \in V$ that belongs to category c_j can be defined by $logP(w_t|c_j)$. These scores are pre-computed in the training stage and accumulated for each word $w_{d_i,k}$, while the content is scanned from the beginning to the end. The computation in Equation 5.4 is very fast because only the addition operation is performed for each word. This is why we select NB as the basis of the early decision algorithm.

5.3.3 Keyword Extraction in The Training Stage

The classifier of the early decision algorithm is trained off-line from sample Web content in both the banned and allowed categories. We used the *Rainbow* program (`www-2.cs.cmu.edu/~mccallum/bow/rainbow`) and its library, *Bow*, to train the classifier. They can extract keywords with high information gain as the features from the training categories. Common words, such as "the", "of" and so on (called stop words), should be dropped from the keyword set because they help little in classification (with low information gain).

Automatic keyword extraction could become complex for Web content in some oriental languages. Unlike in western languages (e.g., English), no space characters delimit the words in these languages. Hence it is unclear how many characters compose a semantic "word", not to mention a keyword.

We suggest the N-gram method [PS03, CL96] for extracting keywords in CJK (Chinese, Japanese and Korean) languages, where N-gram means N characters. The tool to extract keywords, *Rainbow*, is modified to do so. The modified algorithm for the N-gram keyword extraction is detailed in [HL03]. Simply put, the N-gram algorithm looks for keywords of various lengths (i.e., the N-gram) and determines the best length of each keyword.

A simple rule can eliminate redundant keywords in the N-gram extraction:

If (1) a string s is a substring of another string t, and (2) every appearance of s implies the appearance of t, only t is left as the keyword. We prefer leaving only the long keyword (i.e., t) to reduce the false-positive possibility due to short keywords. Eliminating the word s causes no harm because s provides no more clues for classification than t does. The score of a concatenated string is defined to be the maximum score of each composite substring. For instance, $Score(st) = max\{Score(s), Score(t)\}$, where $Score(s)$ denotes the score of string s.

5.3.4 The Filtering Stage

In the filtering stage, the incoming content of a Web page is scanned from the beginning for the extracted keywords. Suppose $n\%$ of the page content has been scanned. The event $E_{n,m}$ denotes that the accumulated score has reached m when the filter has scanned $n\%$ of the content. The probability that this content belongs to a category $c_j \in C$ is derived from

$$P(c_j|E_{n,m}) = \frac{P(E_{n,m}|c_j)P(c_j)}{P(E_{n,m}|c_j)P(c_j) + P(E_{n,m}|c'_j)P(c'_j)}. \quad (5.5)$$

- $E_{n,m}$: the event that when the filter has scanned $n\%$ of the content, the accumulated score has reached m.
- $P(c_j)$: the estimated probability that category c_j appears in typical Web content. $P(c_j)$ can be estimated from sample traffic beforehand or dynamically measured in a running environment by recording and analyzing actual Web content.
- $P(c'_j)$: the estimated probability that category c_j does not appear in typical Web content. $P(c'_j) = 1 - P(c_j)$.
- $P(E_{n,m}|c_j)$: the estimated probability that $E_{n,m}$ happens given that the content belongs to category c_j. The estimate of $P(E_{n,m}|c_j)$ is the number

of Web pages in c_j that $E_{n,m}$ happens divided by the number of Web pages in c_j.

- $P(E_{n,m}|c'_j)$: defined similarly as $P(E_{n,m}|c_j)$, except that c_j is replaced with c'_j.

To accelerate the computation of $P(c_j|E_{n,m})$, two two-dimensional tables of $P(E_{n,m}|c_j)$ and $P(E_{n,m}|c'_j)$ are built for each n and m from the training sets in the training stage, for each $c_j \in C$. Note that the table look-up is a little bit tricky herein. The accumulated score of m in the filtering may not exactly match any subscripts of m on the tables because the subscripts of m on the tables are discrete and finite. Therefore, the probabilities $P(E_{n,m}|c_j)$ and $P(E_{n,m}|c'_j)$ are derived in practice by looking up the tables with n and the maximum subscript of m no larger than the accumulated score of m.

Figure 5.2 presents the pseudo-code of the early decision algorithm. Two thresholds, T_{bypass} and T_{block}, are set arbitrarily to be 0.1 and 0.9 herein. Let PCE_j be the estimate of $P(c_j|E_{n,m})$. If $PCE_j < T_{bypass}$, for all c_j in the list of banned categories, the content is unlikely to be in a banned category, and the remaining content can be bypassed. On the contrary, if there exists some c_j in the list of banned categories such that $PCE_j > T_{block}$, the content is likely to belong to c_j, and should be blocked by the filter. The computation overhead in Equation 5.5 is nearly negligible, occupying less than 0.1% of the total classification time in our profiling.

The classifier should scan a minimum amount of the content in a Web page to avoid deciding too early from only the very front part of the content. If the content in a banned category happens to not have keywords in this part, false negatives may occurs. The parameter *min_scan* in Figure 5.2 denotes the minimum amount in percentage. We set *min_scan=15* arbitrarily since it is sufficient to effectively

avoid false negatives in our experiment.

5.4 Experiments

5.4.1 Performance Metrics

To measure the accuracy of the early decision algorithm, we use the F1 measure that combines the recall and the precision by taking the harmonic average of them with equal weight [Van79]. We also use two metrics: the *average scan rate* (ASR) and the throughput, defined by

$$ASR = \frac{Total\ bytes\ that\ are\ scanned}{Total\ bytes\ in\ the\ content} \times 100\% \tag{5.6}$$

and

$$Throughput = \frac{Total\ bits\ in\ the\ content}{Total\ execution\ time\ (sec)}, \tag{5.7}$$

to measure the effectiveness of acceleration. The former reflects the percentage of the Web content that is scanned in the early decision algorithm, and the latter shows the actual throughput in Web content filtering.

5.4.2 Experimental Results and Discussion

From the experiment, totally 300 sample Web pages in four typically banned categories, Pornography, Game, Online-Shopping and Finance, are randomly collected from the YAHOO directory service, and another 300 pages are from other categories to serve as the allowed content. The early decision algorithm searches the Web content with a multiple-string matching algorithm for the keywords extracted in the training stage. A sub-linear time algorithm (e.g., the Wu-Manber algorithm, which can skip characters in the text by nearly the length of the shortest keywords [WM94]) hardly helps the performance here because short keywords

$Earlybypass \leftarrow False$;

$Earlyblock \leftarrow False$;

$n \leftarrow 0$;

while not end of content **do**

Skip stop words and the HTML tags.

Read the next keyword;

$n \leftarrow$ percentage of the content that has been scanned; {scanning at least min_scan% of content}

if $n > min\_scan$ **then**

$m \leftarrow$ the accumulated score;

for each banned category c_j **do**

$PEC_j \leftarrow P(E_{n,m}|c_j)$ of current scanning position;

$PEC'_j \leftarrow P(E_{n,m}|c'_j)$ of current scanning position;

$PCE_j \leftarrow (PEC_j P(c_j))/(PEC_j P(c_j) + PEC'_j P(c'_j))$;

end for

if $(\forall c_j, PCE_j \leq T_{bypass})$ **then**

$Earlybypass \leftarrow True$;

Exit;

end if

if $(\exists c_j, PCE_j \geq T_{block})$ **then**

$Earlyblock \leftarrow True$;

Exit;

end if

end if

end while

Figure 5.2: The pseudo-code of the early decision algorithm.

Table 5.1: Comparison of classification accuracy in four banned categories.

Category	Original Bayesian classifier			Early decision		
	Pr	Re	F1	Pr	Re	F1
Porn	1.00	.993	.996	.977	.918	.947
Game	1.00	.971	.985	.958	.819	.883
Shopping	1.00	.975	.987	.866	.750	.804
Finance	.896	1.00	.945	.964	.900	.931

are not uncommon in natural languages. The filtering routine is implemented on Lex [LS75], which uses the linear-time Aho-Corasick algorithm [AC75], and thus its performance is independent of the keyword lengths.

The accuracy of the original Bayesian classifier, which scans the entire content, is compared with that of the early decision algorithm for the four banned categories in Table 5.1, in which Pr denotes the precision, Re denotes the recall, and F1 denotes the F1 measure, which is the harmonic average of Pr and Re. Among the categories in comparison, only the shopping category presents noticeable accuracy degradation, while the others remain fairly good accuracy. After a careful examination, we observed that the Web pages in the shopping category have many common words that also appear in allowed categories. Therefore, the score accumulation from keywords is slow. Lacking representative keywords reduces the accuracy if the scanned part is not long enough. We consider the categorization should be more specific in this case so that precise keywords can be extracted.

Table 5.2 presents the average filtering accuracy of the content in the four banned categories (summarized from Table 5.1) and the allowed categories. The accuracy of both types of content with the early decision algorithm is close to that when the entire content is scanned. The speed-up is obvious because the early decision algorithm scans only 17.22% of content in the banned categories

Table 5.2: Average accuracy and scan rate in the early decision algorithm.

Banned			Allowed			ASR	ASR
Pr	Re	F1	Pr	Re	F1	(Banned)	(Allowed)
.941	.847	.892	.947	.920	.934	17.22%	26.51%

Table 5.3: Accuracy in the setting of no false positives in allowed content.

Category	Original Bayesian classifier			Early decision		
	Pr	Re	F1	Pr	Re	F1
Porn	.977	.918	.947	1.00	.773	.871
Game	.958	.819	.883	1.00	.623	.767
Shopping	.866	.750	.804	1.00	.55	.709
Finance	.964	.900	.931	1.00	.730	.843

and 26.51% in the allowed categories on average. A large portion of the Web content is bypassed, and the classification time is significantly shortened.

False positives of allowed content may be considered unacceptable in a practical environment, and a high threshold T_{block} is set. Lifting the threshold T_{block} to 1.0 can effectively avoid false positives in the allowed categories, as shown from the high precision in Table 5.3. Note that lifting T_{block} also leads to more false negatives in the banned categories because some banned content is unable to reach such a high threshold. Therefore, deciding a proper threshold is a tradeoff in practice.

Both the execution time and throughput of the early decision algorithm are compared with those of the original Bayesian classifier to manifest the improvement. Both classifiers are implemented on a PC with Intel Pentium III 700 MHz and 64MB of RAM. Table 5.4 presents the comparison results of filtering both the banned and allowed content. The results show a significant improvement in throughput, about five times higher than that of the original Bayesian classifier for banned content and nearly four times higher for allowed content.

Table 5.4: Comparison of the throughput of the early decision algorithm and the original Bayesian classifier.

Algorithm	Execution time (ms)	Throughput (Mb/s)
Original Bayesian classifier	1333.77	41.05
Early decision for banned content	241.89	226.36
Early decision for allowed content	239.90	156.68

Many commercial products and open source packages in our investigation, such as DansGuardian `dansguardian.org`, can block a page as soon as the score accumulation achieves the given threshold configured arbitrarily by the user. In contrast, the early decision algorithm compares the threshold with the probability estimation of the classification, rather than the score itself. This approach has two advantages over that in DansGuardian. First, the two parameters, T_{bypass} and T_{block}, have stronger association with the accuracy than the threshold on the score in DansGuardian. Therefore, it is easier to customize the thresholds in the early decision algorithm to achieve the desired accuracy. In comparison, deciding a proper threshold in DansGuardian to get the desired accuracy will take more efforts in trial and error, since the threshold provides few clues to the accuracy. Second, the early decision algorithm accelerates not only filtering blocked Web pages, but also filtering allowed pages. The advantage is particularly significant when the Web accesses are mostly allowed content.

The early decision algorithm is also implemented on the content analysis of DansGuardian by modifying its filtering code. In our testing samples, the throughput is about three times higher than that in the original version of Dans-

Guardian. The increasing primarily comes from the better criterion in the content filtering and the acceleration of filtering the allowed content. The principle of early decision can also be implemented into the content filtering process in other Web filtering products.

5.4.3 Practical Consideration in Deployment

With the increasing number of categories to be classified, ambiguity between these categories may increase. In our opinion, the proper place to perform Web content filtering is restricted to the edge devices for performance reason. Such edge devices usually require fewer banned categories, and thus the problem with increasing number of categories is not that serious.

The two thresholds, T_{bypass} and T_{block}, can be tuned according to the tradeoffs between accuracy and efficiency. The accuracy can be increased at the cost of less efficiency by decreasing T_{bypass} or increasing T_{block}, and the efficiency can be increased at the cost of less accuracy by increasing T_{bypass} or decreasing T_{block}. The tuning depends on which is more important for an organization: accuracy or efficiency. For example, if the Web filter is overloaded, it may trade a little accuracy for efficiency to avoid overburdening the system; otherwise, it can adjust the two parameters for better accuracy.

Even though the early decision algorithm significantly speed up the filtering decision, we believe that it should complement other Web filtering approaches, especially URL blocking, but not to replace them. First, URL blocking is faster than content analysis since a URL has much fewer characters to be processed than the Web content. Besides, if a banned URL is successfully blocked, no network bandwidth will be wasted to download the banned content. As discussed in Section 5.2.1, content analysis is still needed to successfully catch the banned

content. The early decision can accelerate this part significantly. Second, Web content may contain images, video, Flash objects, Java applets and so on, which are non-trivial to analyze. Analyzing these objects is beyond the scope of this paper, but it is still helpful to increase the accuracy in filtering the Web content.

In summary, a Web filtering system can support various approaches in practice. The system first blocks URLs according to the database of banned URLs that is constantly maintained. To reduce false negatives due to the outdated database, content analysis can catch the banned content whose source is not in the URL database. The early decision algorithm can speed up content analysis to reduce the latency perceived by the user and to increase the system throughput. Although analyzing other types of objects in the Web content, such as images, could increase the accuracy, it is still a trade-off between performance and processing effort so far. It depends on the user to evaluate whether turning on such an analysis is worthwhile.

5.5 Conclusion

This work addresses the problem with possibly long delay in text classification algorithms that perform run-time content analysis in Web filtering. We present an early decision algorithm to decide to either block or pass the content as early as possible. A significant performance improvement is observed. The throughput is increased by about five times higher for banned content and nearly four times higher for allowed content, while the accuracy remains fairly good. In the F1 measure, the accuracy is about 89% for filtering banned content, and about 93% for allowed content.

The early decision algorithm is simple but effective. The same rationale be-

hind this algorithm can be applied to other content filtering applications as well, such as anti-spam. The algorithm can be combined with more features other than keywords from the text to further increase the overall accuracy of the content filter. Besides, the filtering can be further accelerated by combining the URL-based method with the cached results. That is, by caching the decisions on URLs of the filtered Web pages, duplicate filtering on the same Web page can be avoided. Content analysis can be skipped if the cached URL is matched. The maintenance of the URL database is also facilitated.

CHAPTER 6

Conclusions and future works

To accelerate deep packet inspection, we review existing string matching algorithms, and profile their performance on various DPI applications. From the study, we have two major observations:

1. *Verification frequency*, memory access (thus *cache locality*) and *shift distance* are the three major factors that affect the performance of string matching.

2. String matching on intrusion detection is not so critical as that addressed in the literature because it is common that only part of the traffic, say the HTTP requests, is scanned in practice. Therefore, we focus more on anti-virus since improving string matching is significant to its performance and the size of its pattern set challenges the scalability of a design.

We therefore design a hardware architecture, namely BFAST, and a hybrid algorithm based on the observations. The BFAST architecture exploits *algorithmic heuristic* with Bloom filters to scan the content in sub-linear time, so that the hardware does not sheer rely on hardware parallelism or high frequency for acceleration. The architecture also avoids some practical hurdles with *the bad-block heuristic*, and proposes a method that can achieve linear time in the worst case. The throughput of the design is up to 9.34 Gbps for 16,384 patterns and the block size $b = 8$. In the hybrid algorithm, we separate the pattern set by length

so that the *backward hashing* (BH) algorithm, which is good at long patterns, can scan for only the long patterns. The Aho-Corasick then has only a relatively small set of short patterns, and the reduced automaton improves the performance due to good cache locality. The overall performance is three times faster than the original implementation in ClamAV.

We also propose a probabilistic approach, namely *the early decision algorithm*, to accelerate classification of Web filtering. The algorithm can make the classification decision early before scanning the entire content of both allowed and banned Web pages. The thresholds for passing and blocking a Web pages are also easily configured because they are directly associated with the accuracy. The algorithm increases the throughput by around five times for banned content and nearly four times for allowed content, while the accuracy remains fairly good — about 89% for filtering banned content, and about 93% for allowed content.

There are still some practical issues in the dissertation for further study in the future:

1. The entire packet processing involves not only string matching for DPI. Although the throughput of scanning a buffer could be up to several gigabits per second, the other stages, e.g., loading data into the buffer, could become a bottleneck. A total solution is needed besides accelerating string matching.

2. String matching with an automaton approach is still a bottleneck in software implementation. Although many hardware accelerators can accelerate the automaton tracking significantly, they are not applicable in software implementation, which is unable to take the parallelism in a hardware solution.

3. The packet content in applications such as spam filtering is *structured*, and

the patterns are significant only in a certain context. Therefore, *parsing* the content to learn the contextual information, instead of sheer scanning the text stream for the patterns, should be also accelerated in the future.

References

[AC75] Alfred V. Aho and Margaret J. Corasick. "Efficient string matching: an aid to bibliographic search." *Communications of the ACM*, **18**(6):333–343, June 1975.

[AC07] N. Sertac Artan and H. Jonathan Chao. "TriBiCa: Trie Bitmap Content Analyzer for High-Speed Network Intrusion Detection." In *Proc. of the 26th IEEE Infocom Conference*, Anchorage, AL, May 2007.

[ACF05] Monther Aldwairi, Thomas Conte, and Paul Franzon. "Configurable string matching hardware for speeding up intrusion detection." *ACM SIGARCH Computer Architecture News*, **33**(1):99–107, March 2005.

[ADL04] Michael Attig, Sarang Dharmapurikar, and John Lockwood. "Implementation Results of Bloom Filters for String Matching." In *Proc. 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, Napa Valley, CA, April 2004.

[AGS99] Giuseppe Attardi, Antonio Gulli, and Fabrizio Sebastiani. "Automatic Web page categorization by link and context Analysis." In *Proc. of THAI-99, First European Symp. Telematics, Hypermedia, and Artificial Intelligence*, pp. 105–119, Varese, Italy, 1999.

[BCT06] Benjamin C. Brodie, Ron K. Cytron, and David E. Taylor. "A scalable architecture for high-throughput regular-expression pattern matching." In *Proc. of 33rd International Symposium on Computer Architecture (ISCA)*, pp. 191–202, Boston, MA, July 2006.

[Blo70] Burton H. Bloom. "Space/time tradeoffs in hash coding with allowable errors." *Commun. of the ACM*, **13**(7):422–426, July 1970.

[BM77] Robert S. Boyer and J Strother Moore. "A fast string searching algorithm." *Commun. of the ACM*, **20**(10):762–772, October 1977.

[BMI] *Illustrations of the Boyer-Moore algorithm.*

[BP05] Zachary K. Baker and Viktor K. Prasanna. "A computationally efficient engine for flexible intrusion detection." **13**(10):1179–1189, October 2005.

[BSC06] Joao Bispo, Ioannis Sourdis, Joao M. P. Cardoso, and Stamatis Vassiliadis. "Regular expression matching for reconfigurable packet inspection." In *Proc. IEEE International Conference on Field Programmable Technology (FPT)*, Bangkok, Thailand, December 2006.

[Cav05] Cavium Networks. *OCTEON NSP - network services processor family*, 2005.

[CL96] Hsin-Hsi Chen and Jen-Chang Lee. "Identification and classification of proper nouns in Chinese texts." In *Proc. of 25th European Conference on Information Retrieval Research (ECIR)*, pp. 222–229, Copenhagen, Denmark, August 1996.

[CM05] Young H. Cho and William H. MangioneSmith. "A Pattern Matching Coprocessor for Network Security." In *Proc. of ACM/IEEE Design Automation Conference (DAC)*, pp. 234–239, Anaheim, CA, June 2005.

[CNM02] Young H. Cho, Shiva Navab, and William H. Mangione-Smith. "Specialized hardware for deep network packet filtering." In *Proc. of 12th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 452–461, La Grand Motte, France, September 2002.

[DKS04] Sarang Dharmapurikar, Praveen Krishnamurthy, Todd Sproull, and John W. Lockwood. "Deep packet inspection using parallel Bloom filters." *IEEE Micro*, **24**(1):52–61, January 2004.

[DL05] Sarang Dharmapurikar and John W. Lockwood. "Fast and scalable pattern matching for content filtering." In *Symposium on Architectures for Networking and Communications Systems (ANCS)*, pp. 183–192, Princeton, NJ, October 2005.

[EC05] Ozgun Erdogan and Pei Cao. "Hash-AV: fast virus signature scanning by cache-resident filters." In *Proc. Globecom*, pp. 1767–1772, St. Louis, MO, November 2005.

[Fro06] Jeffery E.F. Froedl. *Mastering Regular Expressions*. O'Reilly, third edition, 2006.

[FV01] Mike Fisk and George Varghese. "Fast content-based packet handling for intrusion detection." Technical Report CS2001-0670, UCSD, 2001.

[Gal79] Zvi Galil. "On improving the worst case running time of the Boyer-Moore string matching algorithm." *Communications of the ACM*, **22**(9):505–508, 1979.

[GGM04] Todd J. Green, Ashish Gupta, Gerome Miklau, Makoto Onizuku, and Dan Suciu. "Processing XML streams with deterministic automata and stream indexes." *ACM Trans. Database Systems*, **29**(4):752–788, December 2004.

[GM01] Pankaj Gupta and Nick McKeown. "Algorithms for packet classification." *IEEE Network*, **15**(2):529–551, March/April 2001.

[GTL02] Eric J. Glover, Kostas Tsioutsiouliklis, Steve Lawrence, David M. Pennock, and Gary W. Flake. "Using Web Structure for classifying and describing Web pages." In *Proc. of World Wide Web (WWW)*, pp. 562 – 569, Honolulu, HI, May 2002.

[HCC06] Mohamed Hamammi, Youssef Chahir, and Liming Chen. "WebGuard: A Web filtering engine combining textual, structural and visual content-based analysis." *IEEE Trans. Knowledge and Data Engineering*, **18**(2):272–284, February 2006.

[HL03] Fu-Hsiang Huang and Ying-Dar Lin. *"Evaluating the accuracy and efficiency of a multi-language content filter."*. Master's thesis, National Chiao Tung University, 2003.

[KA94] Jeffrey O. Kaphart and William C. Arnold. "Automatic extraction of computer virus signatures." In *Proc. of 4th Virus Bulletin of International Conference*, pp. 178–184, Abingdon, England, September 1994.

[KST03] Jari Kytojoki, Leena Salmela, and Jorma Tarhio. "Tuning String Matching for Huge Pattern Sets." In *Proc. of Symposium on Combinatorial Pattern Matching (CPM)*, pp. 211–224, Morelia, Mexico, 2003.

[LHC04] Rong-Tai Liu, Nen-Fu Huang, Chih-Hao Chen, and Chia-Nan Kao. "A fast pattern-match engine for network processor-based network intrusion detection system." In *Proceedings of Information and Technology: Coding and Computing (ITCC)*, pp. 97–101, Las Vegas, NV, April 2004.

[LHF02] Pui Y. Lee, Siu C. Hui, and Alvis Cheuk M. Fong. "Neural networks for Web content filtering." *IEEE Intelligent Systems*, **17**(5):48–57, September/October 2002.

[LJL06] Ying-Dar Lin, Chi-Wei Jan, Po-Ching Lin, and Yuan-Cheng Lai. "Designinig an integrated architecture for network content security gateways." *IEEE Computer*, **39**(11):66–72, November 2006.

[LLL06] Po-Ching Lin, Zhi-Xiang Li, Ying-Dar Lin, Yuan-Cheng Lai, and Frank C. Lin. "Profiling and accelerating string matching algorithms in three network content security applications." *IEEE Commu. Surveys and Tutorials*, **8**(2), Second Quarter 2006.

[LLLar] Po-Ching Lin, Ying-Dar Lin, Yuan-Cheng Lai, and Tsern-Huei Lee. "Using string matching for deep packet inspection." *IEEE Computer*, to appear.

[LS75] Mike Lesk and Eric Schmidt. "Lex — A lexical analyzer generator." Technical Report Comp. Sci. Tech. Rep. No. 39, Bell Laboratories, 1975.

[LTH07] Ying-Dar Lin, Kuo-Kun Tseng, Chen-Chou Hung, and Yuan-Cheng Lai. "Scalable Automaton Matching for High-Speed Deep Content Inspection." In *21th IEEE Advanced Information Networking and Applications (AINA)*, Niagara Falls, Canada, May 2007.

[LTLar] Ying-Dar Lin, Kuo-Kun Tseng, Tseng-Huei Lee, Chen-Chou Hung, and Yuan-Cheng Lai. "A Platform-Based SoC Design and Implementation of Scalable Automaton Matching for Deep Packet Inspection." *Journal of Syst. Arch.*, to appear.

[Lun06] Jan van Lunteren. "High-Performance Pattern-Matching for Intrusion Detection." In *Proc. of the 25th IEEE Infocom Conference*, Barcelona, Spain, April 2006.

[MDW04] Yevgeniy Miretskiy, Abhijith Das, Charles P. Wright, and Erez Zadok. "Avfs: An On-Access Anti-Virus File System." In *USENIX Security Symposium*, pp. 73–88, San Diego, CA, August 2004.

[Mit96] Tom Mitchell. *Machine learning.* McGraw Hill, 1996.

[MLL03] James Moscola, John W. Lockwood, Ronald Loui, and Michael Pachos. "Implementation of a content-scanning module for an Internet firewall." In *Proc. of IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 31–38, Napa Valley, CA, April 2003.

[MM96] Robert Muth and Udi Manber. "Approximate Multiple Strings Search." In *Proc. of Symposium on Combinatorial Pattern Matching (CPM)*, pp. 75–86, Laguna Beach, CA, 1996.

[Nor04] Mark Norton. "Optimizing pattern matching for intrusion detection." Technical report, Sourcefire, Inc., 2004.

[NRa] Marc Norton and Dan Roelker. *Multi-rule inspection engine.* http://www.snort.org/docs.

[NRb] Marc Norton and Dan Roelker. *Snort 2.0 protocol flow analyzer*. http://www.snort.org/docs.

[NR00] Gonzalo Navarro and Mathieu Raffinot. "Fast and flexible string matching by combining bit-parallelism and suffix automata." *ACM Journal of Experimental Algorithms*, **5**(4):1–36, 2000.

[NR02] Gonzalo Navarro and Mathieu Raffinot. *Flexible Pattern Matching in Strings*. Cambridge University Press, 2002.

[NR04] Gonzalo Navarro and Mathieu Raffinot. "New techniques for regular expression searching." *Algorithmica*, **4**(2):89–116, November 2004.

[PAD06] Vern Paxson, Krste Asanović, Sarang Dharmapurikar, John Lockwood, Ruoming Pang, Robin Sommer, and Nicholas Weaver. "Rethinking Hardware Support for Network Analysis and Intrusion Prevention." In *USENIX Workshop on Hot Topics in Security*, pp. 63–68, Vancouver, Canada, August 2006.

[PP05] Giorgos Papadopoulos and Dionisios Pnevmatikatos. "Hashing + memory = low cost, exact pattern matching." In *Proc. of 15th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 39–44, Tampere, Finland, August 2005.

[PS03] Fuchun Peng and Dale Schuurmans. "Combining naive Bayes and n-gram language models for text classification." In *Proc. of 25th European Conference on Information Retrieval Research (ECIR)*, pp. 105–119, Pisa, Italy, April 2003.

[RFB97] M. V. Ramakrishna, E. Fu, and E. Bahcekapili. "Efficient hardware hashing functions for high performance computers." **46**(12):1378–1381, December 1997.

[Seb02] Fabrizio Sebastiani. "Machine learning in automated text categorization." *ACM Computing Survey*, **34**(1):1–47, March 2002.

[SIH04] Yutaka Sugawara, Mary Inaba, and Kei Hiraki. "Over 10 Gbps string matching mechanism for multi-stream packet scanning systems." In *Proc. of 14th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 484–493, Antwerp, Belgium, September 2004.

[SL05] Haoyu Song and John W. Lockwood. "Multi-pattern signature matching for hardware network intrusion detection systems." In *Proc. of the 48th IEEE Globecom Conference*, St. Louis, MO, November 2005.

[SP03] Robin Sommer and Vern Paxson. "Enhancing byte-level network intrusion detection signatures with context." In *Proc. ACM Computer and Communications Security (CCS)*, Washington D.C., October 2003.

[SP04] Ioannis Sourdis and Dionisios Pnevmatikatos. "Pre-decoded CAMs for efficient and high-speed NIDS pattern matching." In *Proc. 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 258–267, Napa Valley, CA, April 2004.

[SPW05] Ioannis Sourdis, Dionisios Pnevmatikatos, Stephan Wong, and Stamatis Vassiliadis. "A reconfigurable perfect-hashing scheme for packet inspection." In *Proc. of 15th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 644–647, Tampere, Finland, August 2005.

[Tar06] Tarari Inc. *Tarari RegEx5 product brief*, 2006.

[TLL05] Kuo-Kun Tseng, Ying-Dar Lin, Tseng-Huei Lee, and Yuan-Cheng Lai. "A Parallel Automaton String Matching with Pre-Hashing and Root-Indexing Techniques for Content Filtering Coprocessor." In *16th IEEE International Conference on Application-Specific Systems, Architectures, and Processors (ASAP)*, Samos, Greece, 2005.

[TLLar] Kuo-Kun Tseng, Yuan-Cheng Lai, Tsern-Huei Lee, and Ying-Dar Lin. "A Fast Scalable Automaton Matching Accelerator for Embedded Content Processors." *ACM Trans. Embedded Comput. Syst.*, to appear.

[TS06] Lin Tan and Timothy Sherwood. "Architectures for bit-split string scanning in intrusion detection." *IEEE Micro*, **26**(1):110–117, January 2006.

[TSC04] Nathan Tuck, Timothy Sherwood, Brad Calder, and George Varghese. "Deterministic memory-efficient string matching algorithms for intrusion detection." In *Proc. of the 23th IEEE Infocom Conference*, pp. 333–340, HongKong, China, March 2004.

[Van79] C. J Van Rijsbergen. *Information Retrieval*. Dept. of Computer Science, University of Glasgow, second edition, 1979.

[Wan] James Z. Wang. *WIPE: Wavelet image pornography elimination*. http://wang.ist.psu.edu/docs/projects/wipe.html.

[Wan01] James Z. Wang. *Integrated region-based image retrieval*. Kluwer Academic Publishers, Dordrecht, Holland, 2001.

[WM94] Sun Wu and Udi Manber. "A fast algorithm for multi-pattern searching." Technical Report TR94-17, Dept. Comput. Sci., Univ. Arizona, 1994.

[WM95] Wm A. Wolf and Sally McKee. "Hitting the memory wall: implications of the obvious." *Computer Architecture News*, **23**(1):20–24, March 1995.

[Xil04] Xilinx Inc. *Two flows for partial reconfiguration: module based and difference based*, September 2004.

[Xil05] Xilinx Inc. *Virtex-II Pro and Virtex-II Pro X platform FPGAs: complete data sheet*, October 2005.

[YCD06] Fang Yu, Zhifeng Chen, Yanlei Diao, T. V. Lakshman, and Randy H. Katz. "Fast and memory-efficient regular expression matching for deep packet inspection." In *Proc. of ACM/IEEE symposium on Architecture for networking and communications systems (ANCS)*, pp. 93–102, San Jose, CA, December 2006.

[YL99] Yiming Yang and Xin Liu. "A re-examination of text categorization methods." In *Proc. of 22nd ACM International Conference on Research and Development in Information Retrieval*, pp. 42–49, Berkeley, CA, August 1999.