doi:10.11959/j.issn.1000-436x.2015215

HashTrie：一种空间高效的多模式串匹配算法

张萍[1,2,3]，刘燕兵[1,3]，于静[1,3]，谭建龙[1,3]
（1. 中国科学院 信息工程研究所，北京 100093；2. 中国科学院大学，北京 100049；
3. 信息内容安全技术国家工程实验室，北京 100093）

摘 要：经典的多模式串匹配算法 AC 的内存开销巨大，已经无法满足当前高速网络环境下大规模特征串实时匹配的应用需求。针对这一问题，提出一种空间高效的多模式串匹配算法—HashTrie。该算法运用递归散列函数，将模式串集合的信息存储在位向量中，以取代状态转移表来减少空间消耗，并利用 Rank 操作进行快速匹配校验。理论分析表明，HashTrie 算法的空间复杂度为 $O(|P|)$，与模式串集合的规模 $|P|$ 线性相关，与字符集大小 σ 无关，优于经典多模式串匹配算法 AC 的空间复杂度 $O(|P|\sigma\log|P|)$。在随机数据集和真实数据集（Snort、ClamAV 和 URL）上的测试结果表明，HashTrie 算法比 AC 算法节约高达 99.6%的存储空间，匹配速度约为 AC 算法的一半左右。HashTrie 算法适合于模式串集合规模较大、模式串长度较短的多模式串匹配问题，是一种空间高效的多模式串匹配算法。

关键词：入侵检测；多模式串匹配；位向量；递归散列函数；空间高效
中图分类号：TN925 **文献标识码**：A

HashTrie: a space-efficient multiple string matching algorithm

ZHANG Ping[1, 2, 3], LIU Yan-bing[1, 3], YU Jing[1, 3], TAN Jian-long[1, 3]
(1.Institute of Information Engineering, Chinese Academy of Sciences, Beijing 100093, China;
2. University of Chinese Academy of Sciences, Beijing 100049, China;
3. National Engineering Laboratory for Information Security Technologies, Beijing 100093, China)

Abstract: The famous multiple string matching algorithm AC consumed huge memory when the string signatures were massive, thus unable to process high speed network traffic efficiently. To solve this problem, a space-efficient multiple string matching algorithm−HashTrie was proposed. This algorithm adopted recursive hash function to store the patterns in bit-vectors in place of the state transition table in order to reduce space consumption. Further more it made use of the rank operation for fast verification. Theoretic analysis shows that the space complexity of HashTrie is $O(|P|)$, which is linear with the size of pattern set $|P|$ and is independent of the alphabetsize σ. The space complexity is superior to the complexity $O(|P|\sigma\log|P|)$ of AC. Experiments on synthetic datasets and real-world datasets(such as Snort, ClamAV and URL) show that HashTrie saves up to 99.6% storage cost compared with AC, and in the meanwhile it runs at a matching speed that is about half of AC. HashTrie is a space-efficient multiple string matching algorithm that is appropriate to search large scale pattern strings with short lengths.

Key words: intrusion detection; multiple string matching;bit-vector; recursive hash function; space-efficient

收稿日期：2014-10-21；**修回日期**：2015-04-10
基金项目：国家自然科学基金青年基金资助项目(61202477)；国家高技术研究发展计划(“863”计划)基金资助项目(2011AA010703)；中国科学院战略性科技先导专项基金资助项目(XDA06030602)
Foundation Items: The National Natural Science Foundation of China (61202477); The National High Technology Research and Development of China (863 Program) (2011AA010703); The Strategic Priority Research Program of the Chinese Academy of Sciences (XDA06030602)

1　引言

在计算机科学中，模式串匹配问题是最古老、也是研究最广泛的问题之一。在发展迅猛的网络安全、信息检索、计算生物学等领域，模式串匹配技术的应用尤为广泛，具体应用包括：网络入侵检测、计算机病毒特征码匹配、搜索引擎、DNA 序列比对等。所谓多模式串匹配，是指从任意的字符序列（文本）中找出一组给定的字符串（模式串）的所有出现位置。

自 20 世纪 70 年代以来，国内外有上百种模式串匹配算法被相继提出。Navarro 和 Raffinot 根据算法的搜索方式不同，将模式串匹配算法分为 3 类[1]：前缀模式、后缀模式和子串模式。这些算法的通用机制是使用一个固定长度的窗口来搜索文本，检验窗口内文本是否匹配，然后将窗口尽可能地向右移动，直至整个文本结束。算法的差别主要体现在如何检验窗口内文本的匹配，以及窗口的移动策略上。传统的、应用比较广泛的多模式串匹配算法有 AC[2]、Multiple Shift-And[3]、BM[4]、Commentz-Walter[5]、Horspool[6]、Wu-Manber[7]、SBDM[8]、SBOM[9]、Multiple BNDM[10]等。

在目前应用最广泛的经典方法中，如 AC 算法，仍然主要采用二维存储结构—状态转移表，来存储状态转移信息，无法避免内存空间开销过大的问题。当前的空间压缩方法主要从如何压缩状态转移自动机和如何构造新的数据结构存储状态转移表 2 个方面入手。Lin[11]等提出了一种基于完美散列函数的存储结构，在内存使用效率上提高明显。但在实际应用中，对完美散列函数的计算代价会极大地影响算法性能。

针对上述问题，本文从数据结构的角度出发，提出一种基于前缀的多模式串匹配算法—HashTrie 算法。理论分析表明，HashTrie 算法的空间复杂度仅为 $O(|P|)$，优于经典多模式串匹配算法 AC 的空间复杂度 $O(|P|\sigma\log|P|)$。实际数据测试表明，相较于经典的模式串匹配算法 AC 和基于双数组结构实现的 AC 算法[12]，HashTrie 算法极大地降低了内存开销。以 Snort 数据集为例，HashTrie 比 AC 节约高达 99.6%的存储空间。此外，在所有测试算法中，HashTrie 算法的预处理时间是最短的，比经典算法 AC 节约了 90%以上的预处理时间，更能满足入侵检测系统对规则生效时间的实时性要求。HashTrie 算法更适合模式串集合规模较大、模式串长度较短的多模式串实时匹配问题。

在本文后续章节中用到的相关符号定义如下：给定模式串集合 $P=\{p^{(0)},p^{(1)},\cdots,p^{(r-1)}\}$，任意的输入文本 $T=t_0t_1\cdots t_{n-1}$。其中 $p^{(i)}$ 是定义在有限字母表（字符集）$\sum$ 上的字符串 $p^{(i)}=p_1^{(i)}p_2^{(i)}p_3^{(i)}\cdots p_{m_i}^{(i)}$。记 P 中最长模式串的长度为 $l_{\max}$，记 $|P|$ 为 P 中所有字符串长度之和，即 $|P|=\sum_{i=0}^{r-1}\left|p^{(i)}\right|=\sum_{i=0}^{r-1}m_i$。

2　相关工作

多模式串匹配算法国内外的代表性工作介绍如下。

Aho 和 Corasick[2]提出了基于前缀搜索的多模式串匹配算法—Aho-Corasick 算法（简称 AC 算法），从模式串集合构建 AC 自动机，通过对自动机的访问进行匹配。该算法匹配的时间复杂度正比于待扫描文本长度，不受关键词长度和文本统计特征的影响，性能比较稳定。但是需要巨大的存储空间来存储自动机，通常不是最快的匹配算法。

Wu 和 Manber[7]提出的基于后缀搜索的方法—Wu-Manber 算法（简称 WM 算法）是 Boyer-Moore 算法的扩展和改进。该算法使用散列函数将所有可能的字符块散列到跳跃距离表 SHIFT 上，然后利用 SHIFT 表进行快速地移动以跳过不可能匹配的文本字符。Wu-Manber 算法简单高效，实际效果好。但该算法适合于字符集比较大、模式串长度比较长的应用场景，不适用于模式串长度比较短的场景。

Erdogan 和 Cao[13]提出的 Hash-AV，作为 WM 的一个变体，使用一组散列函数和适合于 CPU 二级高速缓存的布隆过滤器阵列。Hash-AV 在不访问主存储器情况下，过滤掉多数的“不匹配”情况，在缓存中实现快速扫描。但不可避免地继承了 WM 的缺陷，该算法同样不适用于模式串长度比较短的场景。

Tan 和 Sherwood[14]提出了一种按位拆分的自动机存储结构 bit-split。该方法将一个 AC 有限状态机按位拆分为一组较小的 AC 有限状态机，以减少总内存需求。但按位分割存储结构是一种基于硬件的实现方案，软件实现效率较低。

Van[15]提出了一种基于硬件实现的可编程状态机机制。该方法将模式串集聚类拆分，并将状

态存放在一个 256 行的表中，移除 AC 有限状态机中那些转移到初始状态和初始状态的下一个状态的状态。采用散列函数来减小内存开销，用上界 4 限制状态转移的最大散列冲突数目。但是对散列冲突的处理会增加存储器访问次数，降低算法的性能。

Song[16]等设计了基于缓存的确定性有限自动机的匹配算法 ACC，提出后继状态寻址机制来存储和访问内存中 DFA 的状态转移边。运用此种方法，转移边可以被有效地存储和直接访问。然而在该机制下，仍然使用传统二维存储结构存储有多条状态转移边的状态。当此类状态增加时，算法性能降低。

Lin[11]等采用完美散列函数来压缩状态转移表，以消除散列冲突。相比传统的二维存储结构，算法在性能和内存使用效率上均有所提高。但在实际应用中，计算完美散列函数的代价很大，会极大地影响算法的性能。对完美散列函数值的计算亦受到存储空间的制约，加重了算法在 GPU 上访问内存的开销。

面对当前互联网协议设计缺陷、计算机系统漏洞、网络入侵攻击等日趋严峻的网络安全问题，已有算法的存储空间和运算速度已经难以满足高速网络环境下特征串实时匹配的应用需求。因此，设计更加高效的多模式串匹配算法，具有重要的理论和实际意义。

3 HashTrie：基于递归散列函数的多模式串匹配算法

HashTrie 算法的基本处理流程如图 1 所示。

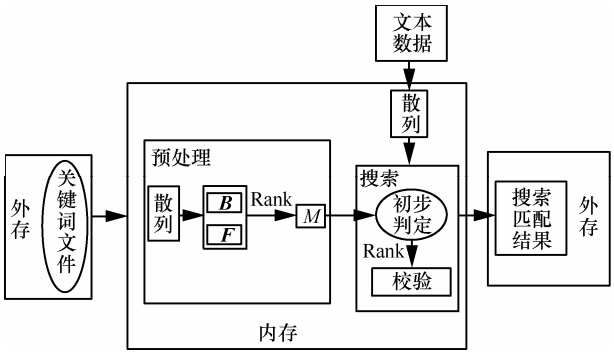

图 1 HashTrie 算法的基本处理流程

读入关键词文件，算法进入预处理阶段，利用递归散列函数和 Rank 操作构建 HashTrie。读入文本数据，进入搜索阶段。利用递归散列函数计算当前字符串的散列值，并结合之前构建的 HashTrie，对文本进行逐个字符搜索、校验。最后，报告最终搜索匹配的结果。

HashTrie 算法主要包含 2 个阶段：预处理阶段和搜索阶段。在预处理阶段，主要任务是构造 HashTrie 所需的数据结构。HashTrie 包含 3 个数据结构：位向量 ***B***、位向量 ***F*** 和校验散列表 *M*。

HashTrie 数据结构的构造过程如图 2 所示。以模式串集合{ she，his，this，sheer}为例，利用递归

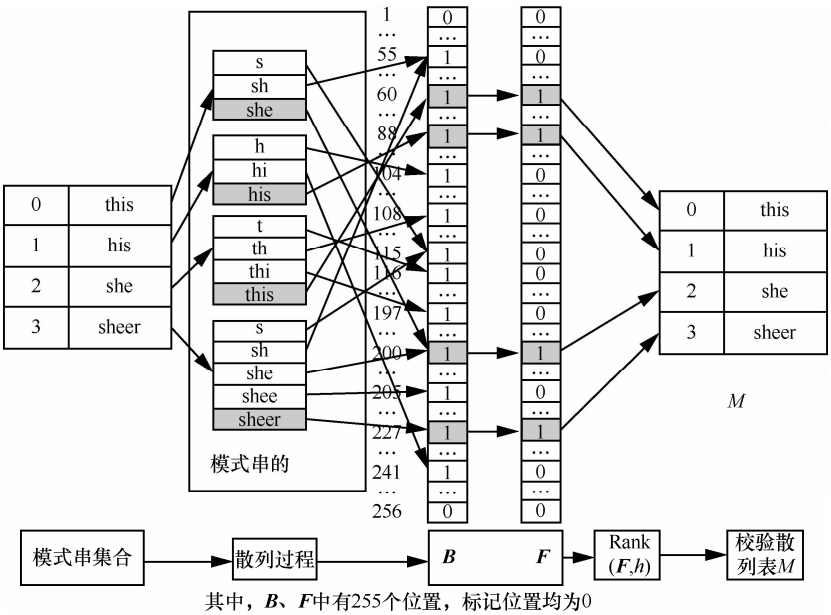

图 2 HashTrie 数据结构构造

散列函数计算模式串前缀的散列值，存储到对应的位向量 ***B*** 和 ***F***。再利用 Rank 操作构建对应的校验散列表 *M*。

3.1　递归散列函数设计和 Rank 操作

在利用递归散列函数构建 HashTrie 的过程中，算法所使用的递归散列函数形式如下。

$$H_m = \mathrm{Hash}(t_1 \cdots t_m) = \left(\sum_{i=1}^{m} t_i a^{m-i}\right) \bmod H \quad (1)$$

其中，*m* 是字符串的长度，*H* 是位向量 ***B*** 和 ***F*** 的大小，H_m 是当前长度为 *m* 的字符串的散列值。该散列函数具有如下递推性质

$$H_m = \left(H_{m-1} a + t_m\right) \bmod H \quad (2)$$

根据上述递推公式，每处理一个新的字符仅需要 *O*(1)的时间计算出新的散列值，因此，采用递归散列函数可以减少计算代价，进而提高算法的效率。在具体实现时，为了保证散列函数的均匀性，上述散列函数中取 $a = 22\,695\,477$，*a* 是 Borland C/C++编译器中伪随机数生成器的参数[17]。这样可以保证所选取的散列函数具有较好的随机性。

此外，在 HashTrie 算法的数据结构构建、搜索和校验过程中，均用到 Rank 操作。Rank 操作是一种快速、高效的位向量查找算法，具体来说，$\mathrm{Rank}(h)$ 计算位向量中前 *h* 个比特中 1 的个数。该方法利用位向量简洁地表示一个集合，并能快速地查找出原集合中的元素。在 Jacobson[18]1989 年的研究工作中，对 Rank 技术进行了详细的描述。概括地说，对于大小为 *H* 的位向量 ***B***，可以在 *O*(1) 时间复杂度内实现对位向量 ***B*** 的 Rank 操作，同时仅需要 *O*(*H*) 的额外存储空间。无论从理论分析还是实验验证，Rank 操作在时间和空间上均是高效的。在 HashTrie 算法的预处理阶段和搜索阶段中，均运用到 Rank 操作，以实现位向量上的查找操作。

3.2　位向量 *B* 和 *F* 的构造

在 HashTrie 构建过程中，首先需要构造 2 个数据结构：位向量 ***B*** 和 ***F***。

对于给定模式串集合 $P = \{p^{(0)}, p^{(1)}, \cdots, p^{(r-1)}\}$，***B***、***F*** 的具体构造过程如下：对每一个模式串 $p^{(k)} = p_1^{(k)} p_2^{(k)} p_3^{(k)} \cdots p_{m_k}^{(k)} \in P$ 的每一个前缀 $u = p_1^{(k)} p_2^{(k)} p_3^{(k)} \cdots p_j^{(k)} \in P$，其中，$0 \leqslant k < r, 1 \leqslant j \leqslant m_k$，利用递归散列函数计算该前缀的散列值 $h = \mathrm{Hash}(u)$，同时将位向量 ***B*** 中第 *h* 位置为 1。依此，模式串的每一个前缀 *u* 经过散列之后映射到位向量 ***B*** 中对应的一个比特，得到位向量 ***B***。

对于每一个完整的模式串，即一个可能匹配，除了在位向量 ***B*** 中标记其散列值外，同时将其标记在另一个位向量 ***F*** 中。即将 ***F*** 中对应的第 *h* 位亦置为 1，得到位向量 ***F***，以此记录完整模式串的信息。其中 ***F*** 的大小与 ***B*** 相同。

算法 1　位向量 ***B***、***F*** 的构造

Construct(***B***、***F*** for $P = \{p^{(0)}, p^{(1)}, \cdots, p^{(r-1)}\}$)

1) $|P| \leftarrow \sum_{p \in P} |p|$
2) $l_{\max} \leftarrow \max_{p \in P} |p|$
3) $H \leftarrow 2^{\lceil \mathrm{lb}16|P| \rceil}$
4) $B[0, H-1] \leftarrow 0^H$
5) $F[0, H-1] \leftarrow 0^H$
6) **For** each pattern $p^{(k)} = p_1^{(k)} p_2^{(k)} p_3^{(k)} \cdots p_{m_k}^{(k)} \in P$
7) $h \leftarrow 0$
8) **For** $j \leftarrow 1, m_k$ **Do**
9) $h \leftarrow \left(ah + p_j^{(k)}\right) \& (H-1)$
10) $B[h] \leftarrow 1$
11) **If** $j = m_k$ **Then**
12) $F[h] \leftarrow 1$
13) **End If**
14) **End For**
15) **End For**

位向量 ***B*** 和 ***F*** 的大小是由模式串集合中所有模式串的前缀总数决定。对于模式串集合 $P = \{p^{(0)}, p^{(1)}, \cdots, p^{(r-1)}\}$，总的前缀数为 $|P|$。在本算法中，取 $H = 16|P|$。这是因为，在散列过程中存在误识别的可能，即字符串 *u* 不是任何模式串的前缀，在位向量 ***B*** 中对应的第 $h = \mathrm{Hash}(u)$ 比特却为 1。字符串 *u* 被误识别的概率由式决定。

$$\Pr\left(B[h(u)] = 1 \mid u\text{不是}P\text{中任何模式串的前缀}\right) \leqslant 1 - \left(1 - \frac{1}{H}\right)^{|P|} \approx 1 - \mathrm{e}^{-\frac{|P|}{H}} \quad (3)$$

取 $H = 16|P|$，$1 - \mathrm{e}^{-\frac{|P|}{H}} = 1 - \mathrm{e}^{-\frac{1}{16}} = 0.06$，即可保证将字符串被误识别的概率控制在 6%之内[19,20]。

此外，将位向量的大小 *H* 向上取整为 2 的整数次幂，这样可以用效率更高的位与操作（$\&(H-1)$）来代替模余操作（$\bmod H$），提高算法的效率。即用下面的散列函数

$$H_m = \left(H_{m-1}a + t_m\right) \& (H-1) \tag{4}$$

来取代式（2）。

HashTrie 算法中位向量 ***B*** 和 ***F*** 的具体构造过程见算法 1。

3.3 校验散列表 *M* 的构造

针对之前 ***B***、***F*** 构造过程中提到的误识别情况，在搜索过程中需要进一步对已识别结果进行校验。在预处理阶段，构造辅助数据结构校验散列表 *M*，借助 *M* 对已识别结果进行校验。

下面介绍校验散列表 *M* 的构造过程。校验散列表 *M* 是一个数组，每个数组元素 *M*[*t*]为一个链表。对于模式串集合 $P=\{p^{(0)},p^{(1)},\cdots,p^{(r-1)}\}$ 中的每一个模式串 $p^{(k)}$，计算该模式串的散列值 *h*，然后利用 Rank 操作计算位向量 ***F*** 中第 *h* 比特在 ***F*** 中的次序 *t*，然后将该模式串存入链表 *M*[*t*]中。在搜索阶段，利用校验散列表 *M* 对已识别结果进行校验，计算出真正的匹配结果。

HashTrie 算法中校验散列表 *M* 的具体构造过程见算法 2。

算法 2 校验散列表 *M* 的构造

Construct (*M* **for** $P=\{p^{(0)},p^{(1)},\cdots,p^{(r-1)}\}$)

1) **For** each pattern $p^{(k)}=p_1^{(k)}p_2^{(k)}p_3^{(k)}\cdots p_{m_k}^{(k)}\in P$ **Do**

2) $h\leftarrow 0$

3) **For** $j\leftarrow 1,m_k$ **Do**

4) $h\leftarrow (ah+p_j^{(k)})\,\&\,(H-1)$

5) **End For**

6) $t\leftarrow \mathrm{Rank}\left(\boldsymbol{F},h\right)$

7) $M[t]\leftarrow M[t]\cup\{p^{(k)}\}$

8) **End For**

3.4 HashTrie 算法的搜索、校验过程

在搜索阶段，主要查找可能匹配的模式串并对其进行校验。从输入文本 *T* 的每个位置 *i* 开始搜索，使用递归散列函数计算文本字符串 $t_it_{i+1}\cdots t_{i+j}$ 的散列值，查找位向量 ***B*** 和 ***F*** 中相应的位置是否为 1，以确定文本字符串 $t_it_{i+1}\cdots t_{i+j}$ 是否可能与某个模式串匹配。若为可能匹配，则进一步进行校验。具体的搜索和校验过程如下。

在搜索过程中，对于文本位置 $i(i=0,\cdots,n-1)$，文本 *T* 的当前搜索窗口为 $t_it_{i+1}\cdots t_{i+j}$。计算当前窗口的散列值 $h=\mathrm{Hash}(t_it_{i+1}\cdots t_{i+j})$，查看位向量 ***B*** 中对应的第 *h* 个位置的值。

1) 如果 $B[h]=0$，表示 $t_it_{i+1}\cdots t_{i+j}$ 不可能与某个模式串的前缀匹配，针对当前文本位置 *i* 的扫描结束。

2) 否则，文本字符串 $t_it_{i+1}\cdots t_{i+j}$ 可能与某个模式串的前缀匹配，需要进一步确认该前缀是否为一个完整匹配。查看位向量 ***F*** 中对应位置的值 $F[h]$，若 $F[h]=1$，表示当前文本字符串 $t_it_{i+1}\cdots t_{i+j}$ 是一个可能的完整匹配结果。鉴于散列冲突的存在，需要对其进行进一步校验。利用 Rank 操作计算散列值 *h* 在位向量 ***F*** 上的次序 *t*，链表 *M*[*t*]中的元素即为可能命中的模式串，将文本字符串 $t_it_{i+1}\cdots t_{i+j}$ 与可能命中的模式串一一比较，以发现真正的匹配结果，校验过程结束。若 $j<l_{\max}$，则继续读入下一个字符，执行上述步骤 1）和 2）；若 $j\geqslant l_{\max}$，则当前文本字符串 $t_it_{i+1}\cdots t_{i+j}$ 不可能与模式串匹配，针对当前文本位置 *i* 的扫描结束。

HashTrie 算法的具体搜索、校验过程见算法 3。

算法 3 HashTrie 算法的搜索、校验过程

Search($T=t_0t_1\cdots t_{n-1}$ **)**

1) **For** $i\leftarrow 0,n-1$ **Do**

2) $h\leftarrow 0$

3) $j\leftarrow 0$

4) **While** $j<l_{\max}$ **Do**

5) $h\leftarrow \left(ah+t_{i+j}\right)\&\left(H-1\right)$

6) **If** $B\left[h\right]=0$ **Then** break

7) **If** $F\left[h\right]=1$ **Then**

8) $t\leftarrow \mathrm{Rank}\left(\boldsymbol{F},h\right)$

9) **For** each pattern $p\in M\left[t\right]$

10) **If** $t_it_{i+1}\cdots t_{i+j}=p$ **Then**

11) **Report Match** $\left(i,p\right)$

12) **End If**

13) **End For**

14) **End If**

15) $j\leftarrow j+1$

16) **End While**

17) **End For**

3.5 空间和时间复杂度分析

下面分析 HashTrie 算法的空间复杂度和时间复杂度。

定理 1 HashTrie 算法的空间复杂度为 $O\left(\left|P\right|\right)$，预处理阶段（算法 1 和算法 2）的时间复杂度为

$O(|P|)$；搜索阶段（算法 3）的最坏时间复杂度为 $O(l_{\max}n)$。

证明

1) 空间复杂度：HashTrie 算法包括 3 个基本的数据结构：位向量 $\boldsymbol{B}$、位向量 $\boldsymbol{F}$ 和校验散列表 M。其中 $\boldsymbol{B}$ 和 $\boldsymbol{F}$ 的大小均为 H，M 用于存储所有待校验的模式串信息，其存储空间大小为 $O(|P|)$，所以

$$\begin{aligned}&|\boldsymbol{B}|+|\boldsymbol{F}|+|M|\\&=H+H+O(|P|)\\&=2\times 2^{\lceil \mathrm{lb}16|P|\rceil}+rO(|P|)\\&=O(N)+O(|P|)\end{aligned}$$

根据算法 1 中对参数 H 的选择：$H=2^{\lceil \mathrm{lb}16|P|\rceil}$，位向量 $\boldsymbol{B}$、位向量 $\boldsymbol{F}$ 和校验散列表 M 总的存储空间为 $O(|P|)$。此外，HashTrie 算法还需要额外的辅助空间以便于在位向量 $\boldsymbol{F}$ 上进行 Rank 操作，其所占用的存储空间为 $\frac{H}{4}=O(|P|)$。综上，HashTrie 算法的空间复杂度为 $O(|P|)$。

2) 预处理时间复杂度：在预处理阶段，需要计算所有模式串的每个前缀的散列值，并在位向量 $\boldsymbol{B}$ 中将相应的比特置 1。对于模式串集合 P，总的前缀数为 $|P|$。通过递归散列函数计算每个模式串前缀的散列值仅需要 $O(1)$ 的时间，因此，构造位向量 $\boldsymbol{B}$ 的时间为 $O(|P|)$。同样地，构造位向量 $\boldsymbol{F}$ 和校验散列表 M 的时间为 $O(|P|)$。此外，还需要在位向量 $\boldsymbol{F}$ 上构造 Rank 操作所需的辅助数据结构，其时间复杂度为 $O(H)=O(|P|)$。因此，HashTrie 算法预处理阶段的时间复杂度为 $O(|P|)$。

3) 搜索时间复杂度：在搜索阶段，对于每个文本位置 i，需要从当前位置开始搜索可能出现的模式串，搜索的最大深度为 $l_{\max}=\max_{p\in P}|p|$。搜索时，每处理一个文本字符（计算散列值）的时间为 $O(1)$。因此，HashTrie 算法在搜索阶段的最坏时间复杂度为 $O(l_{\max}n)$。

表 1 是 HashTrie 算法同经典的 AC 算法[2]、基于双数组结构实现的 AC 算法（简称 DAC）[12]的空间和时间复杂度比较。

在存储空间方面，AC 算法的空间开销主要用于存储状态转移的二维矩阵，因此其空间复杂度与字符集大小 σ 成正比。DAC 算法在 AC 算法的基础上，利用双数组结构表示 Trie 结构，将空间复杂度降为 $O(|P|\log|P|)$，与字符集大小 σ 无关。根据定理 1 的分析可知，本文提出的 HashTrie 算法的空间复杂度 $O(|P|)$，进一步降低了空间复杂度，明显优于 AC 和 DAC 算法。

表 1　HashTrie 算法和 AC、DAC 复杂度对比

算法	空间	预处理时间	搜索时间
AC	$O(\|P\|\sigma\log\|P\|)$	$O(\|P\|\sigma)$	$O(n)$
DAC	$O(\|P\|\log\|P\|)$	$O(\|P\|^2\sigma)$	$O(n)$
HashTrie	$O(\|P\|)$	$O(\|P\|)$	$O(l_{\max}n)$

在预处理时间方面，HashTrie 算法的预处理时间复杂度也是最低的。AC 和 DAC 算法的预处理时间与模式串规模$|P|$和字符集大小 σ 相关，而 HashTrie 仅与模式串规模$|P|$线性相关，与字符集大小 σ 无关。

在搜索时间方面，HashTrie 算法的最坏时间复杂度与最长模式串的长度 $l_{\max}$ 线性相关，高于 AC 算法和 DAC 算法。但是，下一节中的实验结果表明，HashTrie 算法的平均识别长度（算法 3 中变量 j 的平均值）通常是比较小的，远远小于最长模式串长度 $l_{\max}$。因此，在随机情况下，HashTrie 算法的平均搜索时间复杂度可以认为是 $O(n)$。

4　实验评估

本节从存储空间、匹配速度和预处理时间 3 个方面，将 AC 自动机的指针实现方式（AC）、表实现方式（TAC）、双数组实现方式（DAC）和本文提出的 HashTrie 算法进行对比。此外，在随机数据集上对 HashTrie 算法的平均识别长度进行统计，考察模式串长度和平均识别长度的关系，从实验结果上对 HashTrie 算法在搜素阶段的平均时间复杂度进行实验性分析。

实验的软硬件环境如下。CPU：Intel(R) Core(TM) i7-3820 @ 3.60 GHz；内存：32 GB；硬盘：2 TB；操作系统：Ubuntu 12.04.3 LTS；编译环境：GCC Version 4.6.3（Ubuntu/Linaro 4.6.3-1ubuntu5）。

测试数据集包括两部分：开源系统中提取的真实数据集和随机生成的数据集。其中真实数据集包括 MIT 入侵检测数据集[21]、Snort 规则集[22]、ClamAV 规则集[23]、URL 数据集[24]。所采用数据集简要介绍如下。

1) MIT 入侵检测数据集：来自 MIT 公开的网络入侵检测数据集（294.31 MB），用作匹配 Snort 和 ClamAV 规则集的待扫描文本。

2) Snort 规则集：从开源入侵检测系统 Snort 中提取的 6 325 条规则（606.7 KB），作为待匹配的模式串集合。其中最长模式串长度为 366，最短模式串长度为 5。

3) ClamAV 规则集：从开源反病毒系统 ClamAV 中抽取的 79 560 条规则（30.32 MB），作为待匹配的模式串集合。其中最长模式串长度为 233，最短模式串长度为 3。

4) URL 数据集：从网络流量中采集的约 2 000 万（2.08 GB）条 URL 规则作为待扫描文本，从中抽取了 100 万条 URL 规则作为待匹配的模式串集合。其中最长模式串长度为 60，最短模式串长度为 4。

5) 随机数据集：随机生成模式串集合和待扫描文本。模式串和文本中的字符服从等概率独立同分布，生成每个字符的概率为 $\frac{1}{256}$。模式串个数由 100 变化到 100 000，长度为 8，待扫描文本大小为 100 MB。

表 2 是 HashTrie 与 AC、TAC、DAC 等算法在上述数据集上的实验结果。

4.1 存储空间

在存储空间方面，从表 2 中的实验结果可以看出，HashTrie 算法是所有测试算法中占用空间最少的。以 Snort 规则集为例，HashTrie 算法比指针方式实现的 AC 算法节省了 99.6%的存储空间，比表结构方式实现的 AC 算法 TAC 节省了 99.2%的存储空间，比双数组结构方式实现的 AC 算法 DAC 节省了 66.5%的存储空间。

算法的内存空间占用依赖于模式串的个数、长度以及字符集大小等因素。算法所采用的数据结构直接决定了其所占用的内存空间大小。相较于 AC、TAC、DAC，HashTrie 算法在内存空间占用上要远远低于其他 3 种算法，是一种空间高效的多模式串匹配算法。

4.2 匹配速度

在随机数据集上，HashTrie 算法的匹配速度约为 AC、TAC 和 DAC 的一半左右。具体地，在 Snort 规则集上，HashTrie 的匹配速度为上述算法的 42%~87%；在 ClamAV 规则集上，HashTrie 的匹配速度为上述算法的 40%~77%；在 URL 规则集上，HashTrie 的匹配速度为上述算法的 47%~64%。

在随机数据上，HashTrie 算法显著快于 AC 和 TAC 算法，与 DAC 的匹配速度相当。HashTrie 的匹配速度分别为 AC、TAC 和 DAC 的 4.28 倍、4.08 倍和 1.05 倍。

此外，本文还在随机数据集上测试了算法的匹配速度与模式串个数和长度的关系。实验结果如图 3 和图 4 所示。

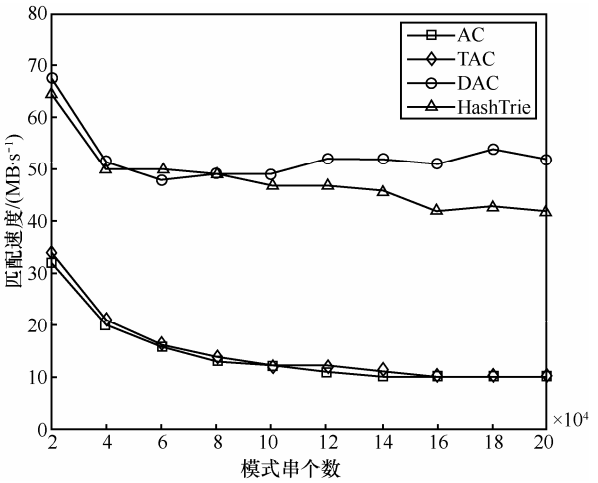

图 3 在随机数据集上(固定 m=8)， AC、TAC、DAC、HashTrie 算法的匹配速度与模式串个数的关系

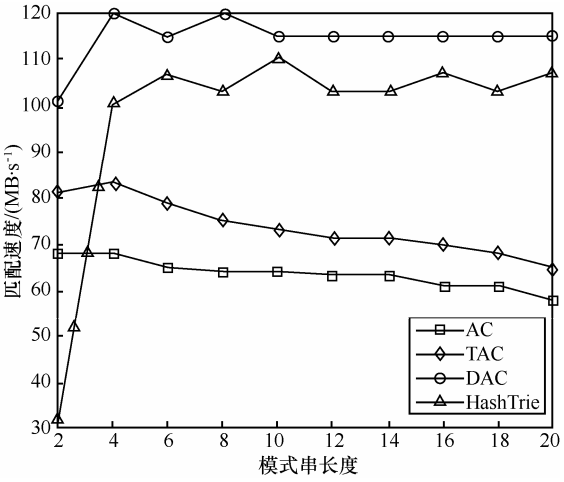

图 4 在随机数据集上(固定 r=5 000)，AC、TAC、DAC、HashTrie 算法的匹配速度与模式串长度的关系

在图 3 中，固定模式串长度 m=8，模式串个数从 20 000 增加到 200 000，考察算法匹配速度随着模式串个数变化的关系。从图 3 中可以看出，随着模式串个数的增加，4 种算法的匹配速度均有所下降。模式串个数在 4 万至 8 万之间，HashTrie 算法的匹配速度高于其他 3 种算法。随着模式串数量的增加，HashTrie 算法的匹配速度高于 AC 和 TAC，

是 AC 和 TAC 的 4 倍。在 10 万至 20 万之间，HashTrie 算法低于 DAC 算法。

在图 4 中，固定模式串个数 r=5 000，模式串长度从 2 变化到 20。从图 4 中可以看出，HashTrie 算法的匹配速度是经典算法 AC 的 1.7 倍左右，相比于 AC 和 ATC 算法，HashTrie 与 DAC 更适合于较长模式串的匹配问题。随着模式串长度变化，HashTrie 算法本身的匹配速度变化相对平稳。

4.3　预处理时间

在实时入侵检测系统中，具有较短预处理时间的串匹配算法更能满足检测规则生效的时效性要求。因此，预处理时间是衡量算法好坏的一个重要指标。

从表 2 中的实验结果可知，HashTrie 算法在所有数据集上的预处理时间均是最短的。HashTrie 比经典的 AC 算法节约 93%的预处理时间，比表方式实现的 TAC 算法节省了 92%的预处理时间，比双数组结构实现的 DAC 算法节省了 86%的预处理时间。因此，HashTrie 算法更能满足实时入侵检测系统对规则生效的时效性要求。

4.4　平均识别长度

最坏情况下，HashTrie 算法的搜索时间复杂度与最长模式串长度 l_{max} 成正比，但是在平均情况下，HashTrie 的搜索效率是比较高的。为了评估 HashTrie 在平均情况下的搜索效率，定义 HashTrie 的平均识别长度为 HashTrie 算法在搜索阶段的平均扫描深度，即扫描过程中在每一个文本位置跳出循环所需扫描的字符串长度（算法 3 中变量 j 的平均值）。实验在随机数据集上进行，固定模式串个数 r=100 000，命中率为 0.001，模式串长度从 4 变化到 256。实验结果如图 5 所示。

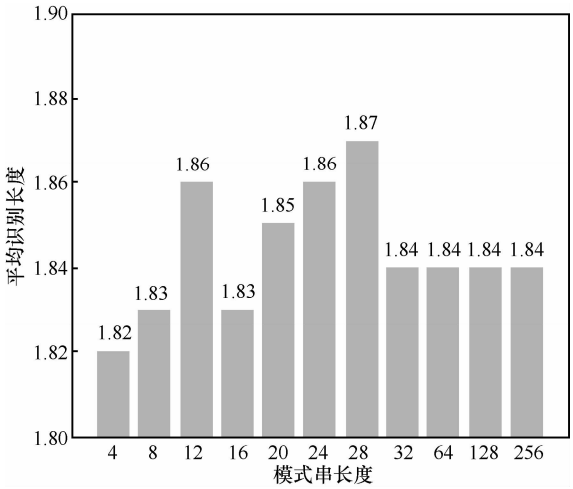

图 5　HashTrie 算法在随机数据集上的平均识别长度

从图 5 可以看出，随着模式串长度从 4 变化到 256，HashTrie 算法的平均识别长度均小于 2，远远小于最坏情况下的识别长度（即最长模式串长度 l_{max}）。因此，在随机情况下，HashTrie 算法的平均

表 2　HashTrie 算法与 AC、TAC、DAC 实验结果对比

数据集	算法	模式串数量	预处理时间/s	内存用量/MB	匹配速度/(MB·s^{-1})	匹配数量
Snort	AC	6 325	0.15	240.686	255.92	98 537
	TAC	6 325	0.12	119.833	181.672	98 537
	DAC	6 325	0.07	2.834	124.181	98 537
	HashTrie	6 325	0.01	0.95	108.202	98 537
ClamAV	AC	79 560	11.44	14 930.818	32.413	18 523 455
	TAC	79 560	8.48	7 427.517	27.976	18 523 455
	DAC	79 560	7.16	158.116	54.3	18 523 455
	HashTrie	79 560	0.14	46.573	21.609	18 523 455
URL	AC	794 615	16.55	22 609.141	109.269	41 591
	TAC	794 615	18.33	11 266.525	79.275	41 591
	DAC	794 615	10.76	289.771	79.783	41 591
	HashTrie	794 615	0.57	201.329	51.031	41 591
随机数据集	AC	100 000	1.02	1 288.529	11.858	10 534
	TAC	100 000	0.76	642.636	12.448	10 534
	DAC	100 000	0.61	19.008	48.387	10 534
	HashTrie	100 000	0.02	9.076	50.847	10 534

搜索时间复杂度可以认为是 $O(n)$，与 AC、TAC 和 DAC 等算法相当。

5 结束语

本文提出了一种基于前缀搜索的多模式串匹配算法 HashTrie，与经典的多模式串匹配算法 AC、TAC 和 DAC 算法相比，HashTrie 大大减少了存储空间消耗，存储空间仅与模式串规模线性关系，与字符集大小 σ 无关。在真实数据集和随机数据集上的测试结果表明，HashTrie 算法比 AC 节约高达 99.6%的内存空间，匹配速度约为 AC 算法的一半至 4 倍。此外，HashTrie 算法在所有数据集上的预处理时间均是最短的，比 AC、TAC 和 DAC 等算法节省了约 90%的预处理时间。HashTrie 更适合模式串集合规模较大、模式串长度较短的多模式串实时匹配问题。下一步将研究在 GPU 上对 HashTrie 算法进行优化的策略以提高其匹配速度。

参考文献：

[1] NAVARRO G, RAFFINOT M. Flexible Pattern Matching in Strings: Practical On-line Search Algorithms for Texts and Biological Sequences[M].Cambridge University Press, 2002.

[2] AHO A V, CORASICK M J. Efficient string matching: an aid to bibliographic search[J]. Communications of the ACM, 1975, 18(6): 333-340.

[3] BAEZA-YATES R, GONNET G H. A new approach to text searching[J]. Communications of the ACM, 1992, 35(10): 74-82.

[4] BOYER R S, MOORE J S. A fast string searching algorithm[J]. Communications of the ACM, 1977, 20(10): 762-772.

[5] COMMENTZ-WALTER B. A String Matching Algorithm Fast on the Average[M]. Springer Berlin Heidelberg, 1979.

[6] HORSPOOL R N. Practical fast searching in strings[J]. Software: Practice and Experience, 1980, 10(6): 501-506.

[7] WU S, MANBERU.A fast algorithm for multi-pattern searching[R]. Technical Report TR-94-17, University of Arizona, 1994.

[8] RAFFINOT M. On the multi backward dawg matching algorithm (MultiBDM)[A]. Proc of 4th South American Workshop on String Processing[C]. 1997. 149-165.

[9] ALLAUZEN C, CROCHEMORE M, RAFFINOT M. Factor oracle: a new structure for pattern matching[A]. SOFSEM'99: Theory and Practice of Informatics[C]. Springer Berlin Heidelberg, 1999. 295-310.

[10] NAVARRO G, RAFFINOT M. A bit-parallel approach to suffix automata: fast extended string matching[A]. Combinatorial Pattern Matching[C]. Springer Berlin Heidelberg, 1998. 14-33.

[11] LIN C H, LIU C H, CHANG S C, *et al*. Memory-efficient pattern matching architectures using perfect hashing on graphic processing units[A]. INFOCOM, 2012 Proceedings IEEE[C]. IEEE, 2012. 1978-1986.

[12] AOE J I. An efficient implementation of static string pattern matching machines[J]. IEEE Transactions on Software Engineering, 1989, 15(8): 1010-1016.

[13] ERDOGAN O, CAO P. Hash-AV: fast virus signature scanning by cache-resident filters[J]. International Journal of Security and Networks, 2007, 2(1): 50-59.

[14] TAN L, SHERWOOD T. A high throughput string matching architecture for intrusion detection and prevention[J]. ACMSIGARCH Computer Architecture News, IEEE Computer Society, 2005, 33(2): 112-122.

[15] VAN L J. High-performance pattern matching for intrusion detection[A]. INFOCOM, 2006 Proceedings IEEE[C]. IEEE, 2006. 1-13.

[16] SONG T, ZHANG W, WANG D, *et al*. A memory efficient multiple pattern matching architecture for network security[A]. INFOCOM 2008, The 27th Conference on Computer Communications[C]. IEEE, 2008.

[17] Available online [EB/OL]. http://en.wikipedia.org/wiki/Linear_congruential_generator.

[18] JACOBSON G. Space-efficient static trees and graphs[A]. Foundations of Computer Science[C]. 1989. 549-554.

[19] 何慧敏,刘燕兵,谭建龙,等. 一种基于子串识别的多模式串匹配算法[J]. 计算机应用与软件, 2012, 28(11): 10-14.
HE H M, LIU Y B, TAN J L, *et al*. A substring recognition based multiple patterns string matching algorithm[J]. Computer Applications and Software, 2012, 28(11): 10-14.

[20] LIU Y B, LIU Q Y, LIU P, *et al*. A factor-searching-based multiple string matching algorithm for intrusion detection[A]. Communications (ICC), 2014 IEEE International Conference[C]. IEEE, 2014. 653-658.

[21] Available online[EB/OL]. http://www.ll.mit.edu/IST/ideval/.

[22] Available online[EB/OL]. http://www.snort.org/.

[23] Available online[EB/OL]. http://www.clamav.org/.

[24] Available online[EB/OL]. http://urlblacklist.com/.

作者简介：

张萍（1987-），女，河南唐河人，中国科学院博士生，主要研究方向为网络与信息安全、内容过滤等。

刘燕兵（1981-），男，湖北麻城人，博士，中国科学院副研究员，主要研究方向为文本算法设计、图数据管理与挖掘、网络流识别与处理等。

于静（1989-），女，满族，河北省承德人，中国科学院研究实习员，主要研究方向为图数据模式匹配、计算机视觉等。

谭建龙（1974-），男，湖南长沙人，博士，中国科学院研究员、博士生导师，主要研究方向为网络数据流管理、算法设计、海量正则表达式匹配、图像匹配算法等。