特征匹配引擎设计与实现①

王恩海 (中国科学院计算机网络信息中心 北京 100190；中国科学院 研究生院 北京 100049)

摘 要： 网络入侵防御系统是维护网络安全的重要工具之一。特征匹配引擎是网络入侵防御系统的计算核心，用于从网络包中搜索出已知网络攻击的特征数据。对于现有的网络入侵防御系统，特征匹配引擎在整个系统中占据很大比例的计算时间。特征匹配属于计算密集型应用，对于底层的计算能力具有很高的要求。提出了一种并行特征匹配算法，并充分利用底层硬件特征，将提出的算法映射到现有的多核处理器上。在 IBM System x3455 上实现的系统具有高达 17.2Gbps 的处理速度和 5 万字条的特征字典容量，其处理速度和特征字典容量超过现有文献中的结果，包括 FPGA、ASIC、网络处理器以及 GPU 上的实现。

关键词： 网络入侵防御系统；模式匹配；网络安全

Design and Implementation of Pattern Matching Engine

WANG En-Hai (Computer Network Information Center, Chinese Academy of Sciences, Beijing 100190, China; Graduate University, the Chinese Academy of Sciences, Beijing 100049, China)

Abstract: Network Intrusion Prevention System (NIPS) is one of the effective tools in providing network security. The core computing function of an NIPS is Pattern Matching Engine (PME), which is used to search pattern data of a known network intrusion from network packages. In current NIPS, PME consumes a significant portion of the computing time. PME is a computing consuming application, requiring high level performance from the system's base computing power. This article proposes a parallel pattern matching approach and maps the computation onto the existing multi-core CPU by fully utilizing the computing power of the base hardware structure of the prevention system. The implementation of the proposed approach on IBM System x3455 server shows that it provides a typical processing speed of 17.2 Gbps with a capacity of 50,000 pattern signatures, which has exceeded the results of all current documentation, including FPGA, ASIC, network CPU, and GPU.

Keywords: intrusion prevention system; pattern matching; network security

1 引言

网络入侵防御系统（NIPS: Network Intrusion Prevention System）可以甄别利用合法的包头穿过防火墙的网络包攻击，NIPS 的计算基于深度包检测（DPI: Deep Packet Inspection）技术，不仅检查网络包头，而且在网络净荷中搜索可能的攻击特征。在服务器、存储设备、边缘和核心路由器等基础设施中，广泛采用 NIPS 进行网络监控，以保护网络和计算设施免遭恶意攻击。特征匹配引擎是 NIPS 的计算核心，用于从网络包中搜索出已知网络攻击的特征数据。对于现有的 NIPS，比如 Snort，特征匹配引擎在整个系统中占据 60 % 的计算时间[1]。特征匹配属于计算密集型应用，通用微处理器对一个网络包进行特征匹配需要约 3，000 条指令，相对的，轻量级网络处理如路由

① 收稿时间:2010-01-20;收到修改稿时间:2010-03-01

和 IP 转换则只需要约 100 条指令。

特征匹配引擎的功能，是针对给定的特征字典，从网络包中搜索出字典中包含的特征字条。目前公开发表的文献中对特征匹配引擎的研究，针对的特征字典大部分具有几千到一万个特征字条，比如 Snort 的特征字典容量是 6000 左右（2009 年 12 月），本文介绍的工作则是针对一个大得多的字典，其容量达到 5 万字条。从公开文献上看，不同的实现在处理速度和字典容量上具有不同的结果，但是没有见到字典容量超过 2 万条的实现。

特征匹配引擎的工作方式可以根据其处理结果分为两大类，包括精确匹配方法如 Boyer-Moore[2], Aho-Corasick[3],Commentz-Walter[4],Wu-Manber [5]等，以及非精确匹配方法如[6,7]。精确匹配方法的普遍思想是利用一个大的数据表（或者树）进行处理，在字典容量大规模增加时，处理速度下降很快。基于 Bloom 过滤器[8,6]的非精确匹配方法可以应对字典容量大规模增加的情况，其问题是需要一个精确匹配引擎做后续处理。

我们对大容量字典进行的分析发现，经过精巧设计的 Bloom 过滤器在面对大容量字典时可以将计算时间保持在需要的水平，后续精确匹配引擎对 Bloom 过滤器的输出结果进行处理，可以达到比较好的处理速度。本文提出的匹配方法，利用轻量级的 Bloom 过滤器进行前端过滤，保持一定的误报率（False positive ratio），然后利用精确匹配引擎处理前端的输出，最终得到精确匹配结果。在我们的算法设计中，同样考虑到的是底层硬件的存储系统的特征和处理器的 SIMD 能力。在 IBM System x3455 系统上的实现结果显示，我们提出的方法在针对 5 万字条的特征字典时，能够提供 17.2Gbps 的处理速度。

本文的主要贡献包括：(1)提出了一种并行特征匹配算法，并充分利用大容量字典的特征和底层硬件特征；(2)将提出的算法映射到现有的多核处理器上；(3)实现的系统具有比较高的处理速度和字典容量，其处理速度超过现有文献中的结果，包括 FPGA、ASIC、网络处理器以及 GPU 上的实现。

本文下面的部分组织如下：首先在第 2 节讨论特征匹配的相关研究工作，然后在第 3 节分析讨论我们提出的并行算法。第 4 节展示在 Intel 多核处理器上的实现结果，并与现有文献的结果进行比较，第 5 节是结论。

2　现有模式匹配算法分析

模式匹配引擎的功能表示如下：给定文本输入 T=t0, t1, …, tn, 以及有限集合 P = { p1, p2, …, pr}，模式匹配引擎从 T 中找出 T 的子串 Ti，使得：

$$Ti = Pj, \text{ 其中 } 1 \le j \le r$$

对于 NIPS 而言，模式匹配引擎检查网络包净荷部分，找出特征字典中包含的特征字条。现有文献中的模式匹配算法可以按照字条个数分为单字条匹配和多字条匹配算法。Boyer-Moore[2]算法是标准的模式匹配算法之一，它每次搜索一个特征字条，主要用于文本处理程序，比如 Word 的搜索和替代操作。多字条匹配算法每次搜索多个特征字条，Aho-Corasick (AC)[3], Commentz-Walter[4], Wu-Manber[5], Knuth-Morris-Pratt[9], Rabin-Karp[10]等属于多字条匹配算法。

Aho-Corasick (AC)[3], Commentz-Walter[4], Knuth-Morris-Pratt[9]算法的工作原理，是从特征字典构建一个有限状态机，然后利用有限状态机对目标文本进行一遍处理。这类基于有限状态机的算法对内存容量的要求比较高，比较适合小规模的特征字典，在特征容量快速增加时，状态机数据结构对内存的需求非常大。

另外一类解决方案利用哈希算法，将需要处理的数据映射到小的空间，比如 Rabin-Karp[10] 和 Wu-Manber[5]算法。这类方案对内存有一定的需求，但是处理时间比基于状态机的方案要快很多。

非精确匹配算法[6,7]利用特殊的哈希结构从目标文本中找出可能的匹配结果，具有一定的误报率，其中，Bloom 过滤器[6,8]在非精确匹配算法中被广泛应用。Bloom 过滤器实际上是一种用于表达哈希值的压缩结构，具有比较小的误报率和零漏报率。

3　新的并行算法

我们提出的并行匹配算法从处理流程上可以分为两个阶段：前端是一个轻量级的 Bloom 过滤器，用于找出可能存在的匹配，具有一定的误报率；后端是一个精确匹配引擎用以确认前端的过滤结果。

Bloom 过滤器和精确匹配引擎的设计策略如下：

(1) 为了满足高速处理速度的要求，前端的

Bloom 过滤器必须是一个轻量级的计算结构，具有比较低的误报率；

(2) Bloom 过滤器和精确匹配引擎的设计要利用底层硬件特性所提供的计算优势。就我们采用的 IBM System x3455 计算机系统而言，可以充分利用该系统提供的多层存储结构和并行计算结构的优势；

(3) 通常的 Bloom 过滤器用于反映一个特征字条是否属于一个特征字典的一部分，但是，为了充分利用 Bloom 过滤器的计算结果，需要将 Bloom 过滤器的中间计算信息暴露出来，并且传递给后端的精确匹配引擎使用。

3.1 Bloom 过滤器

如图 1 所示，前端的 Bloom 过滤器检查目标文本，并找出可能的匹配字条。与通常的 Bloom 过滤器不同，为了给后端的精确匹配引擎提供信息，新 Bloom 过滤器的输出包括文本本身、特征字条出现位置、字条长度，以及哈希值。在 Bloom 过滤器中，我们提出采用串行连在一起的三个哈希结构。

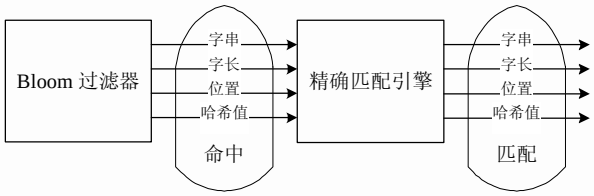

图 1 并行模式匹配算法

如图 2 所示，第一个哈希结构称作双字节位置检测(DBPC)，其数据结构包括一个具有 64K 个元素的向量表，每个向量包含 3 个比特。对于特征字典中的字条，以相邻两个字节为索引，分别映射到 DBPC 表中的一个比特位。DBPC 的哈希值计算是最快的，用于 Bloom 过滤器中的第一遍检测。

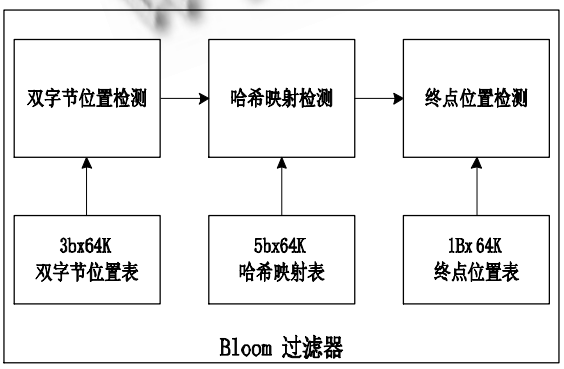

图 2 前端 Bloom 过滤器

第二个哈希结构称作哈希映射检测(HMC)，其数据结构包括一个具有 64K 个元素的向量表，每个向量包括 5 个比特。哈希索引的计算定义如下：对于特征字典中的每个字条，C(i)表示字条的第 i 个字符，Hash(i)表示第 i 个哈希值，则定义：

```
Hash(0) = 0                    /*设置初始值*/
Hash(i+1) =                    /*计算下一个哈希值*/
    ((hash(i) << 2) | (hash(i) >> 14))
    ^ (C(i)<<8 | C(i+1))
    ^ ((hash(i) >> 1)|(hash(i) << 15))
```

第三个哈希结构称作终点位置检测(EPC)，同样采用 64K 的向量表，每个向量 8 比特。该向量表的索引与 HMC 使用同样的哈希值。EPC 用于表示一个位置是否可能是一个特征字条的终结位置。

从三个哈希结构的定义可以看出，DBPC 和 HMC 同时考虑到了输入字符的哈希值和位置，HMC 计算出来的哈希值同样用于 EPC 阶段和后端的精确匹配引擎。EPC 则给出了可能字条的长度。

3.2 精确匹配引擎

精确匹配引擎位于后端，用以验证前端的输出是否真正是一个匹配。精确匹配引擎将特征字典中的字条分成更小的子集合，每一个可能的匹配字条被首先映射到一个这样的子集合，其索引值就是在前面 HMC 阶段计算出来的哈希值，然后在子集合中完成精确匹配。精确匹配引擎包括两个相连的结构：子集合加载和子集合精确匹配。子集合被连续放置在内存(字条数组)中，每个子集合的第一个特征字条的索引值存放在一个称为字条索引表(SRT: String Reference Table)的数据结构中。在计算中，当目标字条的哈希值计算出来之后，该哈希值指向某个子集合，该子集合中的全部字条将被加载到计算单元中，执行子集合精确匹配。

在现在的计算机系统中，用于存放所有子集合的字条数组通常无法全部放置在片上存储器，比如 Cache 中，而是存放在内存中。在多核、多进程的计算环境中，内存的带宽和延迟都无法保证提供可靠的性能。这样，如果字条数组是放置在内存中的，那么会带来精确匹配阶段计算时间上的不确定性。

字条数组访问时间以及子集合容量大小所带来的不确定性，促使我们设计三个新的数据结构，使得精确匹配引擎能够更好的利用字条数组的数据局部性，以提高处理速度。如图 3 所示，字条访问列表

(SAL: String Access List)和字条组(ASG: Assembled String Group)用于存放动态生成的字条子集合访问序列，以及从内存中加载的字条子集合。另外，字条引用表中的每个元素包括两个域，分别指定字条子集合的索引值和子集合中字条的个数。字条子集合在字条表中顺序排放，每个子集合的第一个字条的索引值代表着该子集合，并存放在字条引用表中。字条引用表的容量为 32K 个元素，而字条表存放所有的特征字条。

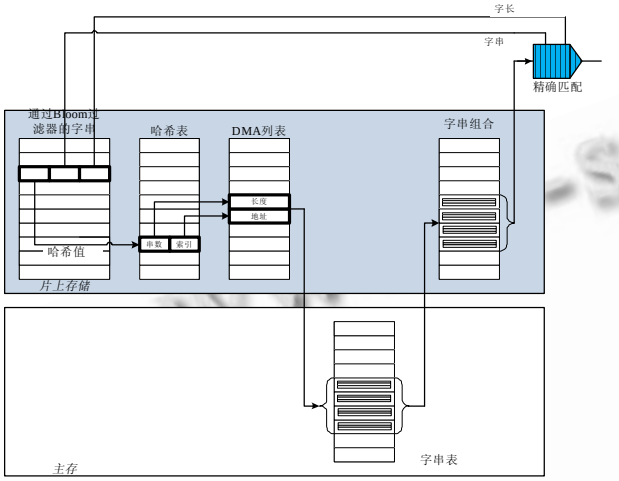

图 3　精确匹配引擎

3.2.1 字条子集合的组合

对于从前端 Bloom 过滤器输出的可能匹配，根据其哈希值可以从字条访问表 SRT 中找到相应的字条子集合，包括子集合的索引值和该子集合中字条的个数。根据这些信息，可以在字条访问表中生产一个 DMA 访问项，每个访问项包括访问基地址和数据长度。同时在字条组 ASG 中生产一个匹配项，包含的域包括：字条本身、字条长度以及字条在原输入文本中的位置信息以备后面引用。

3.2.2 字条子集合的 DMA 加载

在积累足够多的 DMA 访问项之后，可以启动一个 DMA 批处理，将子集合从内存中加载进来。

3.2.3 成组精确匹配

DMA 成组加载完成后，开始进行成组精确匹配操作。字条组 ASG 中每个项的字条，与加载进来的相对应的子集合内的所有字条进行精确匹配。该部分工作用 x86 处理器的单指令流多数据流(SIMD)指令 SSE 进行优化，每个周期可以进行多个字条的精确匹配。

4　实现和性能比较

针对前面给出的并行算法，我们在 IBM System x3455 系统上进行实现。该计算机系统包括两片 AMD Opteron 2000 系列双核处理器，主频 2.8GHz，4G DDR2 667 MHz 内存。我们的实现利用所有的 4 个核进行运算，操作系统是 CentOS 5.0，编译器为 GCC 3.3。

如图 4 所示为我们的并行匹配引擎运行在 1～6 个软件线程上时的性能结果，当针对的字典容量为 5 万字条时，获得最高 17.2Gbps 的处理速度。我们看到，当软件线程为 4 时，系统给出了最高处理带宽；但是软件线程为 3 时的性能已经达到最高性能的 86 %。这是因为，对共享硬件模块，包括内存、总线、高速缓存的访问冲突限制了处理带宽的进一步提高。

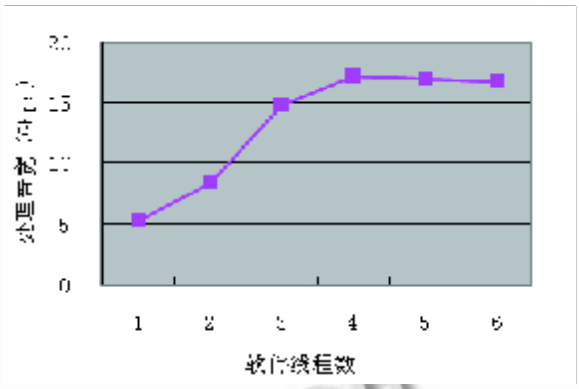

图 4　精确匹配性能

将我们的实现结果与现有文献中的结果进行比较，如表 1 所示，目前可见的文献中所用的特征字条集合的容量都比较小。我们的实现在所有这些实现中的字条容量是其他实现的 2～10 倍，其处理速度和特征字典容量均超过现有文献中的结果，包括 FPGA、ASIC、网络处理器以及 GPU 上的实现

表 1　性能比较

现有实现	带宽 字典容量	平台	标注
[11]	~5Gbps 2,733 条	NePSim2	NP 模拟器上的模拟结果
[12]	4Gbps 1,400 条	Virtex-II	模拟结果
	~16Gbps 1,400 条	0.13 um ASIC	

[13]	~10Gbps 1,000 条	0.13 um ASIC	IC 工艺上的 估计结果
[14]	~10Gbps 2,259 条		FPGA 上的 估计结果
[15]	< 0.1Gbps 6,000 条	NVidia 6800 GT	GPU 上的 软件实现
[16]	3.3~4.4 Gbps 8,400~20, 000 条	Cell B. E.	

5 结论

目前采用有限状态自动机进行特征匹配引擎的设计非常普遍，但是受限于内存容量和计算能力而无法处理大容量的特征字典。采用 FPGA 和 ASIC 可以根据需求进行比较大规模的定制设计，但是目前对成本和时间的要求比较大。本文提出的并行匹配算法，在商用计算系统上实现，设计的算法针对底层硬件特征进行优化，具有非常好的性能以及性价比。本文的实现具有目前已知最快的处理速度和最大的特征字典容量。

参考文献

1 Snort. [2009-12-01]. http://www.snort.org/.

2 Boyer RS, Moore JS. A Fast String Searching Algorithm. Communications of the ACM, 1977,20 (10):762－772.

3 Aho AV, Corasick MJ. Efficient String Matching: An Aid to Bibliographic Search. Communications of the ACM, 1975,18(6):333－340.

4 Commentz-Walter B. A String Matching Algorithm Fast on the Average. Proceedings of the 6th Colloquium, on Automata, Languages and Programming, 1979:118－132.

5 Wu S, Manber U. A fast algorithm for multi-pattern searching. Technical Report, TR 94-17, University of Arizona at Tuscon, 1994.

6 Dharmapurikar S, Krishnamurthy P, Sproull T, Lockwood J. Deep Packet Inspection Using Parallel Bloom Filters: (HOTI'03): Hot Interconnects 11, 2003.

7 Ramaswamy R, Kencl L, Iannaccone G. Approximate fingerprinting to accelerate pattern matching. Internet Measurement Conference 2006:301－306.

8 Bloom BH. Space Time Trade-offs in Hash Coding with Allowable Errors. Communications of ACM, 1970, 13(7):422-426.

9 Knuth D, Morris JH, Pratt V. Fast Pattern Matching in Strings. SIAM Journal on Computing, 1977,6(2):323－350.

10 Karp RM, Rabin MO. Efficient Randomized Pattern-Matching Algorithms. IBM Journal of Research and Development, 1987,31(2):249－260.

11 Piyachon P, Luo Y. Efficient Memory Utilization on Network Processors for Deep Packet Inspection. Proceedings of the 2006 ACM/IEEE symposium on Architecture for networking and communications systems, San Jose, California, 2006:71－80.

12 Brodie BC, Cytron RK, Taylor DE. A Scalable Architecture for High-Throughput Regular-Expression Pattern Matching. Proceedings of ISCA 2006.

13 Tan L, Sherwood T. A high throughput string matching architecture for intrusion detection and prevention. In ISCA'05: 32nd Annual International Symposium on Computer Architecture, 2005:112－122.

14 Dharmapurikar S, Lockwood JW. Fast and scalable pattern matching for content filtering. ANCS 2005: 183－192.

15 Jacob N, Brodley CE. Offloading IDS Computation to the GPU. Proceedings of the 22nd Annual Computer Security Applications Conference on Annual Computer Security Applications Conference (ACSAC 2006), 2006: 371－380.

16 Scarpazza DP, Villa O, Petrini F. High-Speed String Searching against Large Dictionaries on the Cell/B.E. Processor, 22nd IEEE International Parallel & Distributed Processing Symposium (IPDPS 2008), Miami, FL, April 2008.