Computer Science and Application 计算机科学与应用, 2020, 10(9), 1595-1608
Published Online September 2020 in Hans. http://www.hanspub.org/journal/csa
https://doi.org/10.12677/csa.2020.109168

Hans汉斯

针对HTTP压缩流量的基于跳跃的多模式匹配算法

田　源，蒋志颀

中国公安部第一研究所，北京
Email: stevenstian@163.com, jzqtw@foxmail.com

收稿日期：2020年9月1日；录用日期：2020年9月11日；发布日期：2020年9月18日

摘　要

多模式匹配算法是许多网络安全应用中的一种关键核心技术，被应用于检测网络内容中的有害信息。在当前网络中，HTTP压缩技术被广泛应用于网页内容压缩，以提升网络的传输速度。对于压缩HTTP流量的检测，传统方法一般是先对其进行解压，然后利用多模式匹配算法对解压后的内容进行检测过滤。这种传统的方法效率不高，且未能充分利用压缩数据的特性。本文提出了一种针对压缩的HTTP流量的基于跳跃的多模式匹配算法SMCH，SMCH可直接在压缩的HTTP流量上执行多模式匹配操作，而无需额外的解压操作。SMCH可以显著提高针对压缩HTTP流量的匹配速度。实验结果显示，SMCH可直接跳过91.9%的字符，而无需进行字符串匹配操作，其匹配性能比原始的匹配算法相比，提高了将近441%。此外，SMCH算法也更简单，比其它压缩HTTP流量匹配算法具有更高的跳变率和加速比。同时，SMCH具有良好的可扩展性，可以简便地与不同的字符串匹配算法结合使用。在本文中，我们在SMCH上实现了Wu-Manber算法。

关键词

多模式匹配，压缩的HTTP，GZIP，压缩匹配，网络安全

Skipping-Based Multi-Patterns Matching Algorithm for Compressed HTTP Traffic

Yuan Tian, Zhiqi Jiang

First Research Institute of the Ministry of Public Security of PRC, Beijing
Email: stevenstian@163.com, jzqtw@foxmail.com

Received: Sep. 1st, 2020; accepted: Sep. 11th, 2020; published: Sep. 18th, 2020

文章引用: 田源, 蒋志颀. 针对 HTTP 压缩流量的基于跳跃的多模式匹配算法[J]. 计算机科学与应用, 2020, 10(9): 1595-1608. DOI: 10.12677/csa.2020.109168

Abstract

Multi-patterns matching algorithm is a core technique of many network security applications, which is used to detect malicious information of HTTP contents. In the current Internet, HTTP compression technology has been widely used in web content compression to increase transmission speed. For the compressed HTTP traffic, the traditional approach needs to decompress it firstly, and then applies multi-patterns matching algorithm in the decompressed HTTP traffic. This traditional approach is not efficient enough and does not take full advantage of the characteristics of compressed data. In this paper, we propose a novel Skipping-based Multi-patterns Matching Algorithm for Compressed HTTP traffic (SMCH), which can directly perform the multi-patterns matching task on the compressed HTTP traffic without additional decompression. SMCH can significantly accelerate the matching speed of compressed HTTP traffic. Experimental results show that SMCH can skip at most 91.6% of the characters without executing matching, and gain performance boosts most 441% as compared to the original multi-patterns matching algorithm. Besides, SMCH is simpler and has higher skip ratio and accelerate ratio than the other compressed matching algorithms for compressed HTTP traffic. Meanwhile, SMCH has good scalability. Different multi-patterns algorithms can be simply implemented for SMCH. In this paper, we implement Wu-Manber algorithm based on SMCH.

Keywords

Multi-Patterns Matching, Compressed HTTP, GZIP, Compressed Matching, Network Security

1. 引言

模式串匹配技术是入侵检测系统与网络应用防火墙等网络安全应用的关键技术之一，但是随着信息时代的发展，模式匹配技术面临新的问题：数据量急剧增加，网络流复杂多变，匹配的实时性不能满足等。如图 1 所示，展示了自 2010 年以来中国国际出口带宽。所以面对复杂多变的网络数据，模式匹配技术的性能需要进一步提升。

为了不断提升网络带宽，除了提高硬件传输能力以外，压缩数据传输逐渐被重视起来。HTTP 压缩是指将 Web 文本数据在服务器压缩后发送给浏览器的传输方式。目前，Yahoo!、Google、YouTube、Baidu、腾讯等知名网站都使用 HTTP 压缩传输的传输方式。在 2017 年，根据 Alexa 网站[1]的前 1000 万网址的流量统计，有 71.4%的网站使用了 HTTP 压缩传输，目前仍然呈现上升趋势[2]。

在网页内容中存在着大量相同的标签信息以及声明信息等(如 HTML、CSS、JavaScript 等)，所以 HTTP 网页能被有效压缩。根据统计测试，压缩之后的 HTTP 页面可节省将近 70%的存储空间。在 HTTP 1.1 协议中选用的压缩算法包括 GZIP、DEFLATE 等，其中根据[2]可以看出，GZIP 压缩算法是 HTTP 压缩的常用算法，占到了 98.9%左右。

现有的多模式匹配算法过滤压缩数据是比较困难的。针对 HTTP 压缩数据，不做处理和拒绝接受都会带来安全隐患或者性能损耗。目前网络安全应用基本采用“解压 + 扫描”的方式对压缩数据进行扫描过滤，当收到一个 HTTP 数据包之后，首先将其解压得到明文内容，然后利用模式串匹配算法对解压后

的明文进行扫描过滤。这种方式的处理性能不够高效，且存在着一定的安全风险：

1) 处理性能不够高效：这种方式将扫描过滤的过程分解成了数据解压和数据扫描两个彼此独立的阶段，而忽视了压缩数据本身的特性，从而降低了扫描过滤的实时性；

2) 存在着“解压炸弹”的风险：攻击者可故意构造恶意数据来攻击网络安全应用，比如构造 10 GB 内容全为数字 1 的明文数据，但是压缩之后大小仅为 1 MB，甚至更小。当处理这种类型的 HTTP 压缩数据时，解压过程中将会占用大量的内存空间和 CPU 运算，出现“解压炸弹”的风险。

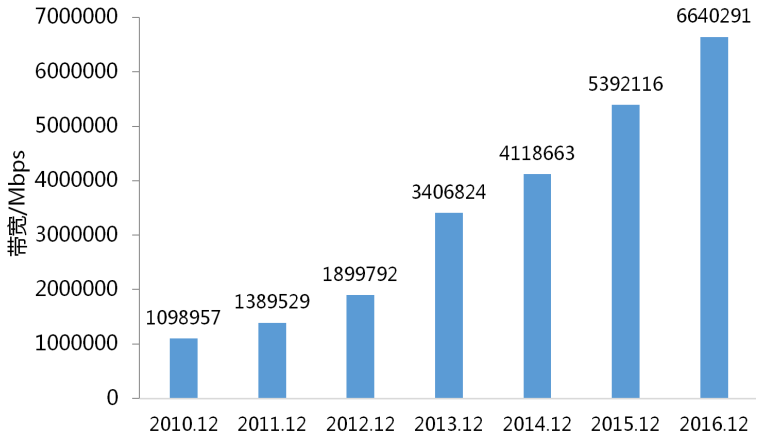

Figure 1. International internet outgoing bandwidth from china since 2010
图 1. 自 2010 年以来中国国际出口带宽

压缩匹配是解决这一问题的有效途径，压缩匹配是指针对压缩数据进行模式匹配时，通过不解压或者局部解压的方式，对压缩数据进行模式串匹配。压缩匹配算法通过利用压缩匹配的特点提高对压缩数据的匹配效率。如图 2 是针对 HTTP 压缩流量整体过滤流程，首先获取 HTTP 报文，根据 HTTP 头部的 Content-Encoding 字段判断是否使用 GZIP 或者 Deflate 编码；如果否，使用正常系统过滤，否则将数据进行哈夫曼解码，获取 LZ77 数据，最后对 LZ77 数据进行过滤，产生过滤结果。

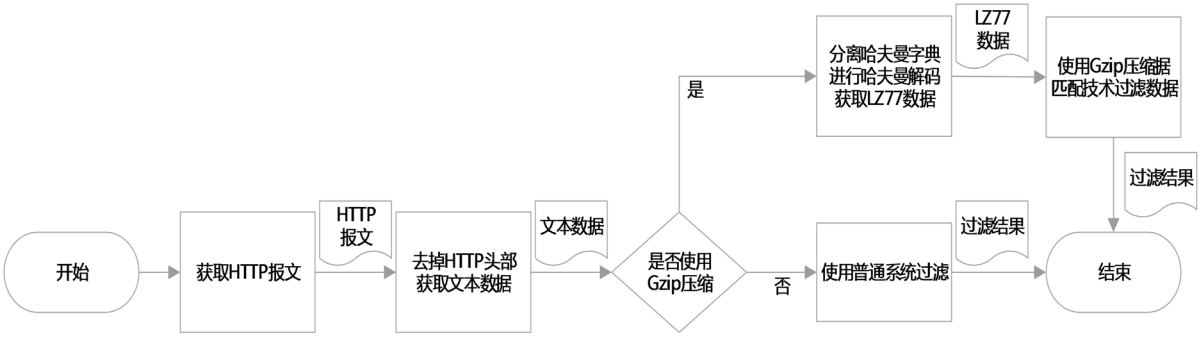

Figure 2. Filter procedure of compressed HTTP data
图 2. HTTP 压缩流量过滤流程

通过对压缩 HTTP 数据的特点进行分析，本文提出了一种针对压缩 HTTP 数据的基于跳跃的多模式串匹配算法 SMCH。SMCH 算法不依赖于任何具体的多模式匹配算法，而可以基于 SMCH 实现不同的支持压缩匹配的多模式匹配算法。在本文中，我们在 SMCH 上实现的是 Wu-Manber 算法。实验结果显示，SMCH 可直接跳过 91.9%的字符而无需进行字符串匹配操作，其匹配性能比原始的匹配算法相比提高了将近 441%。

2. 背景和相关工作

2.1. GZIP 压缩

GZIP，全称是 GNU zip，是一种文件压缩程序，作者是 Jean-loup Gailly 和 Mark Adler，于 1992 年公开发表[3]。GZIP 是对 DELATE 的封装，包括两个部分：LZ77 压缩[4]和哈夫曼编码压缩[5]。由于其中 LZ77 是其主要部分，而且 LZ77 部分是对原始数据进行编码，而哈夫曼编码部分是二次编码，所以本文主要根据 LZ77 的特性对多模式匹配算法进行改进，从而提高匹配速度。

LZ77 算法是一种无损压缩算法，该算法需要维持一个固定大小的窗口(大小一般为 32 KB)。如果字符串在其前面的窗口内重复出现，则使用指针(length, distance)来记录此重复数据，其中 length 表示重复数据的长度，distance 表示重复数据的位置，从而达到压缩数据的目的。例如，字符串“abcdabc”，使用 LZ77 压缩后为“abcd(3,4)”，其中(3,4)表示在此位置有一个长度为 3 的字符串，该字符串的初始位置为向前 4 个单位。

Table 1. LZ77 data info
表 1. LZ77 数据详细信息

总大小/Byte	52,512,646
ASCII 字符大小/Byte	3,281,673
ASCII 字符占比	6.25%
指针的数量	2,536,236
指针的平均长度	19.4

如上所示，LZ77 压缩数据格式中包含两种类型的数据：一种是普通的 ASCII 字符，另一种则是指针(length, distance)，后者表示在当前位置存在重复子串。我们抓取了采用 LZ77 压缩的 Yahoo 网页数据，并对压缩的网页数据进行了统计分析，结果如表 1 所示。压缩的 Yahoo 网页数据中 93.75%的内容是指针，且指针的平均长度为 19.4，这也表明指针部分的数据中有很大部分可以前面重复数据的匹配信息而直接跳过这部分数据而无需执行串匹配操作，从而改善匹配的效率。

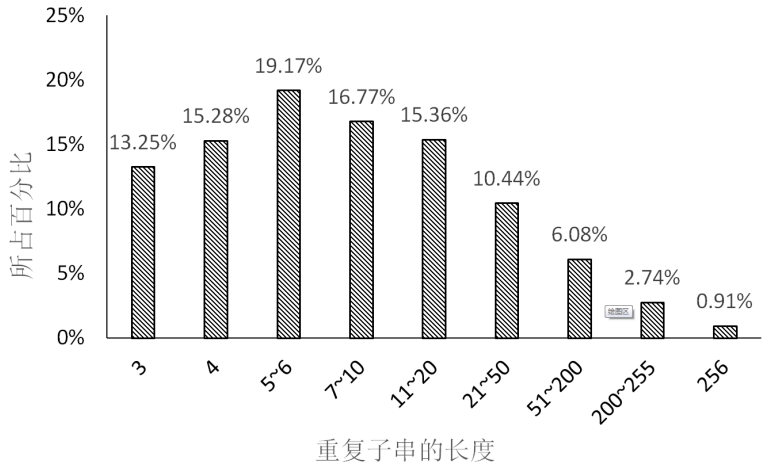

Figure 3. Length distribution of replicated sub-string in LZ77
图 3. LZ77 数据中重复子串的长度分布

如图 3 所示，LZ77 压缩数据中存在着大量的重复子串，如果预先记录下重复子串的匹配信息，就可

以利用前面获得的重复子串的匹配信息来确定当前子串的匹配动作，用重复子串的匹配结果作为当前字符的匹配结果，从而跳过部分字符而无需逐字符的执行串匹配操作。HTTP 数据中存在大量的重复的标签数据使得其有很高的压缩率，这也说明针对压缩的 HTTP 数量，传统的“解压 + 匹配”不是一种高效的处理方式，而是压缩匹配算法是一种更加高效的方法。

2.2. 多模式匹配算法

本模板可直接用于论文及其文字的编排，有的页边距、行距、字体都严格符合规定，请勿修改！尤其是页边距，由于期刊在后期制作过程中需要在页眉、页脚添加各种信息，所以所有论文务必确保现有的页边距不被修改，页面空白不被占用。

目前有许多经典的多模式串匹配算法，如 KMP 算法[6]、AC 算法[7]、Karp Rabin (KR)算法[8] [9]以及 Wu-Manber (WM)算法[10]。但是本文所提的 SMCH 算法不依赖于具体的匹配算法，且无需原始匹配算法的基础匹配结构。本文选择了将 WM 算法和 SMCH 结合使用，并以此为例对 SMCH 进行介绍，因此首先我们先对 WM 算法进行简要的介绍。

WM 可以看作是单模式串匹配算法 BM [11]在多模式串上的扩展。WM 采用了字符块技术，增大了主串和模式串不匹配的可能性，从而增加了直接跳跃的机会。同时使用散列表选择模式串集合中的一个子集与当前文本进行完全匹配，其主要包括三个表，分别是 SHIFT 表、HSAH 表以及 PREFIX 表。SHIFT 表是所有模式串的字符块以及字符块可转移长度的表格，用于在扫描文本串的时候，根据读入的目标文本决定可以转移的字符数，如果相应的转移值为 0，则说明可能匹配成功。PREFIX 表用于存储尾块字符散列值相同的模式串的首块字符散列值，如果转移值为 0，那么就需要详细地将匹配文本和模式串比较。PREFIX 表是为了通过字符块精准的定位模式串的位置，提高比较的效率。HASH 表用来存储尾块字符散列值相同的模式串，待便于精准匹配。

2.3. 压缩匹配算法

压缩匹配算法指的是可以直接在压缩数据上执行字符串匹配的一类匹配算法，可大致分为哈夫曼压缩匹配算法、SDCH 压缩匹配算法、LZW 压缩匹配算法以及 LZ77 压缩匹配算法等。

哈夫曼编码被广泛使用，但是针对哈夫曼编码的压缩匹配算法主要是针对单模式串的。文献[12]总结了哈夫曼编码压缩算法并提出了一种直接在哈夫曼编码上执行匹配的压缩匹配算法。谷歌公司于 2008 年提出了一种新的针对 HTTP 网页的字典类压缩技术 SDCH，文献[13]提出了一种针对 SDCH 压缩技术的压缩匹配算法，该算法主要是基于 AC 算法，假设模式串为{E, BE, BD, BCD, BCAA, CDBCAB}，搜索文本为“ABDDBEAAAACDBCABABCAACBCDBADBC”经过 SDCH 压缩后的文本。算法主要思想为通过不同的压缩数据完成自动机状态的转化，实现跳跃，进而加速，实验结果显示该算法能够跳跃 99%的字符，获得 56%的性能提升。LZW 压缩算法主要用于 GIF、TIFF 以及 PDF 等文件的压缩，目前针对 LZW 压缩数据的压缩匹配算法主要集中在单模式串匹配，相关工作见文献[14] [15]，由于和本文工作关系不大，本文不作详细介绍。

文献[16]首次提出了一种针对压缩 HTTP 流量的压缩匹配算法 ACCH，该算法结合 LZ77 压缩数据的特点，对 AC 算法进行改进，从而实现了直接在压缩的 LZ77 数据上的匹配。ACCH 主要基于匹配状态 Status 和自动机深度 Depth 来实现字符跳跃，文中实验结果显示 ACCH 能够跳跃 75%的字符，且性能较原始的 AC 算法提高了 70%左右。文献[17]以 Wu-Manber 算法为基础，提出了一个针对 LZ77 压缩数据的模式串匹配算法 SPC。SPC 算法能够跳跃最多 87.5%的字符，相较于原始 Wu-Manber 算法，匹配性能提升了 113%，并且相较于 ACCH 算法，也能够达到 52%的性能提升。同时，在额外的

空间消耗方面，SPC 算法也同样优于 ACCH 算法。随后文献[18] [19]分别提出了各自的改进策略，主要针对 AC 算法和 WM 算法进行改进，在一定程度上提升模式串匹配算法在 LZ77 压缩数据上的匹配速度。

LZ77 压缩匹配算法是用于匹配过滤 GZIP 压缩格式的 HTTP 数据的一种有效的方法。但是目前已有的研究工作都是针对某个特定匹配算法进行专门的设计优化，通用性不够。一方面，这些针对性的优化改进算法一般都可以跳跃较多的字符，也能较好的提升算法的匹配性能；但是另一方面，在实际应用中，网络安全应用中一般都集成了多种不同的匹配算法以应对日益复杂的网络内容，不同的匹配算法适用于不同的业务和场景，对多种不同的匹配算法进行优化改进以支持压缩匹配会有较大的难度。针对该问题，本文提出了一种灵活的压缩匹配结构 SMCH，其可以灵活的和不同的具体的匹配算法结合使用。

3. 针对 HTTP 压缩流量的基于跳跃的多模式匹配算法

SMCH 的核心是本文中定义的匹配状态 mState，匹配状态被设计成压缩匹配算法和原始匹配算法之间的桥梁。匹配状态的值由原始匹配算法在匹配阶段计算得到，然后压缩匹配算法根据匹配状态以确定当前字符是否可以被跳过而无需进行字符串匹配操作。

3.1. 匹配状态的定义

除了一些众所周知的英文缩写，如 IP、CPU、FDA，所有的英文缩写在文中第一次出现时都应该给出其全称。文章标题中尽量避免使用生僻的英文缩写。

压缩匹配算法的主要思想是利用压缩数据的性质，记录匹配状态，并利用重复的状态跳跃一定的字符，从而加速过滤。我们可以注意到，压缩匹配算法的重点在于匹配状态。在 ACCH 算法中，CDepth 作为匹配状态，从而根据 CDepth 和 AC 自动机中的状态作为一个比较，判断是否可以跳跃。在 SPC 算法中，Partial Match 作为匹配状态，从而判断是否可以跳跃。匹配状态是压缩匹配算法的重点，压缩匹配算法需要根据匹配状态和搜索的状态进行跳跃。

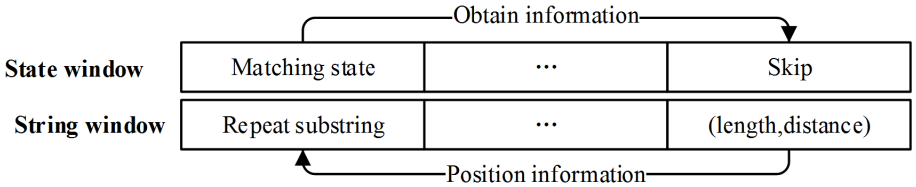

Figure 4. Diagram of compress matching
图 4. 压缩匹配原理示意

如图 4 所示是压缩匹配算法的原理示意图，状态窗口负责记录匹配状态，字符窗口负责缓存字符，以便指针(length, distance)能够索引的正确的子串位置。在遇到指针类型的数据时，我们不仅能够获取原来的子串，同时根据状态窗口可以获取子串对应的匹配状态。通过记录匹配状态和搜索算法的状态判断是否可以重复利用记录的匹配状态完成跳跃。跳跃可能是一段，也可能是一个字符。

在本算法中，将匹配状态表示为 mState，其主要包含两方面的信息：符号和数值。符号表示匹配结果，如果符号为正，那么表示该位置有一个模式串匹配成功；如果符号为负，那么表示该位置在一定长度内不可能匹配成功。数值表示安全距离，即只有大于 mState 的数值时，才是安全的。同时数值在符号为正时，表示匹配成功的模式串长度。当符号为负时，表示在该数值长度内不可能有一个模式串匹配成功。如果数值为 0，则表示该位置没有任何匹配状态。

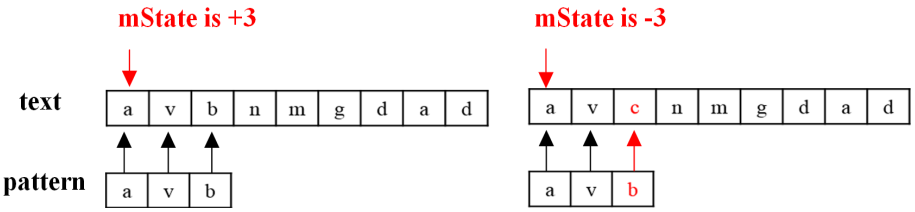

Figure 5. Illustration of mState
图 5. 匹配状态 mState 举例

mState 代表匹配结果，也代表匹配过程。符号表示匹配结果，而数值表示匹配过程。mState 应该在字符串的匹配过程中产生，并不会影响匹配过程。如图 5 所示，图中模式串是“avb”。左侧文本是“avbnmgdad”，右侧文本为“avcnmgdad”。当使用模式串“avb”搜索目标文本“avbnmgdad”时，可以在文本的位置 1 处匹配成功，那么根据 mState 的定义，在第一个位置的字符“a”处的 mState 为“+3”。因为此时匹配成功，所以符号为正。为数值为匹成功的模式串长度，为 3。所以 mState 为“+3”。当目标文本改为“avcnmgdad”时，模式串在第一个位置开始匹配时，在位置 3 的字符‘c’处匹配不成功，而前两个字符“ad”匹配成功。根据 mState 的定义，如果匹配不成功，那么符号应该为负，所以 mState 的符号为负。数值表示安全长度，即大于长度后不可能匹配成功，那么根据图中所示，当大于 3 个字符时，匹配就不可能成功，所以数值为 3。因此，右侧的搜索产生的 mState 为“−3”。

从上面例子可以看出，搜索过程一定有两种可能，一个是匹配成功，一个匹配不成功。当匹配成功时，那么 mState 为正，且数值为匹配成功的模式串长度。当匹配不成功时，mState 可能为负，且数值为最大匹配长度加一，即最小的安全长度。值得注意的是，匹配成功一定可以返回 mState，而匹配不成功时，并不一定返回 mState，主要原因是安全长度的定义。不同的多模式匹配算法有不同的搜索过程，最重要的是保持 mState 的数值必须是安全的，而匹配不成功导致的最大匹配长度加一并不一定是安全的。

在 SMCH 中，跳跃和搜索是分离的。搜索过程是执行字符串匹配并返回当前位置的匹配状态信息，跳跃过程则是根据对数据进行逐个扫描并根据匹配状态进行跳跃。

3.2. 匹配状态的计算

前文已介绍过当前的压缩匹配算法都是对原始匹配算法进行特定的改进优化，但是本文所提的 SMCH 不依赖于原始的匹配算法，具有良好的可扩展性，使用是只需原始匹配算法返回匹配状态即可。此外，SMCH 在获得匹配状态的时候并不需要改变原始匹配算法的基础数据结构，这也使得 SMCH 具备良好的灵活性。

算法 1 字符串匹配算法与 SMCH 的搜索接口

```
1:    procedure Search(String)
2:    Search the String, find the mState.
3:    if matched then
4:        Record the matching information, and return the mState.
5:    else
6:        Return the mState.
7:    end if
8:    end procedure
```

我们在算法 1 中定义 Search()接口作为为模式的搜索过程，它需满足两个条件：搜索字符串的能力和返回正确的 mState 的能力。不同模式匹配算法之间的搜索过程不同，因此重要的是找到搜索与 mState

之间的联系，并将匹配状态转换为 mState。

在本节中，我们仅以 WM 算法的一种实现为例。根据 WM 算法和 mState 的定义，有两个主要计算 mState 的地方：在 shift 表中的转移和在 hash 表中的查询。在 shift 表中，设 max_pat_len 是模式的最大长度，min_pat_len 是模式的最小长度。算法的输入是字符串，其长度是 max_pat_len，从该字符串的某个位置 i 开始，输出 mState 或 mState 数组。在第 4 行，如果 shift_value = 0，则第 5~14 行处理 hash 表中的查询，该查询定位 hash 表并将模式与目标字符串一一比较。在第 8 行，如果文本与模式匹配，则返回模式的长度。否则，最大匹配长度记录在第 10~11 行。max_tmp 是字符串和当前模式之间的最大匹配长度，max 记录的是最大的 max_tmp。在第 14 行，如果 max+1 大于或等于 min_pat_len，则返回−max−1，否则返回−min_pat_len。第 16 行是转移过程，将 mState 设置为[−min_pat_len,−(min_pat_len-shift_value+1)]。这意味着如果一个 shift_value 值不为 0，则有 shift_value 个 mStates 是有效的。

算法 2 SMCH 中的 WM 搜索过程

```
1:    procedure Search(String_{i:i+max_pat_len-1})
2:      shift_value ← shift_table[String_{i+min_pat_len-B:i+min_pat_len-1}]
3:      max ← 0
4:      if shift_value = 0 then
5:        for curPat ← hash_table[Stringi:i+min_pat_len-1] to NULL do
6:          if string ← curPat then
7:            There is a matching, Callback
8:            return curPat.length
9:          else
10:           max_tmp ← the maximal matching length between String and curPat
11:           max ← max_tmp > max ? max_tmp : max
12:         end if
13:       end for
14:       return max+1 ≥ min_pat_len ? -max-1 : −min_pat_len
15:     else
16:       return [−min_pat_len : −min_pat_len+shirt_value-1]
17:     end if
18:   end procedure
```

shift 和 hash 之间有一些区别。在 hash 中成功匹配时，匹配的模式长度与 mState 有关。若有许多可匹配的模式，则返回可匹配模式的最小长度，因为它足够作为安全长度。当哈希失败时，存在一个与某模式可匹配的最大长度(如 max)，这意味着从最大长度开始的下一个字符没有成功匹配。所以 max + 1 是安全长度。另外，max + 1 应该大于或等于 min_pat_len，因为需要将 min_pat_len 长的字符串作为哈希表的索引。但是在 shift 中，min_pat_len 与 mState 显然无关。从当前位置 i 到 i+shift_value-1 的位置是有效的 mState，其值是从−min_pat_len 到−(min_pat_len-shift_value+1)。例如，如果 shift_value 为 3，min_pat_len 为 4，则从当前位置 i 到 i+2 的 mStates 为[−4, −3, −2]。因为所有的转移字符都是安全的，不可能有匹配。

从 hash 表中获得的 mState 仅在此位置有效，而从 shift 表中获得的 mState 表可能会影响之后 shift_value 的位置。主要原因是哈希值的位置不能影响下一个位置，所以不能确定下一个位置的 mState。shift 表是转移表，因此它会对移位的字符产生影响。

SMCH 算法需要一个模式算法来提供 mState，它并不会改变原有模式匹配算法的数据结构，因此模式匹配算法的空间复杂度没有改变。SMCH 算法主要基于两个固定长度的窗口(GZIP 中为 32 KB)，一个是字符的缓存，另一个是 mState 的缓存。不需要额外的空间。

Search()的实现多种多样，但原理相同。在实践中我们还根据 KR 算法和 AC 算法实现了接口，同样可达到良好的应用效果。本文不多叙述其他模式算法的实现。

3.3. 匹配状态在压缩匹配算法中的应用

根据 mState 的定义，跳转的条件是剩余重复子串的长度大于或等于 mState 的绝对值。在本节中，我们将详细介绍如何使用 mState 进行跳转。

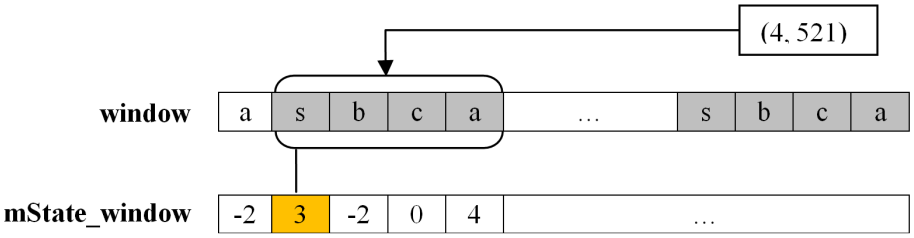

Figure 6. The skip procedure in algorithm SMCH
图 6. SMCH 算法中的跳转过程

首先我们举例说明如何使用 mState，如图 6 所示。指针(4, 521)指向重复子串 sbca，在重复子串中与 s 对应位置的 mState 为 3。可以看出，从 s 开始的重复子串的剩余长度为 4，大于安全长度 3，说明 sbc 段是安全的。换句话说，在 sbc 中有一个匹配项，之前由 mState 记录过。因此，我们不需要搜索字符串 sbc，可以跳过它。可以看到 b 字符的 mState 为−2，根据 mState 的定义，表示第二个字符不能匹配成功，只有 1 个部分匹配成功。但是 b 之后的重复子串的剩余长度是 3 (bca 的长度是 3)，因此我们可以安全地得出结论：bca 字符串不会成功匹配，那么同样可以不搜索而直接跳转。但是，c 不能跳过，因为 0 表示这里没有状态，a 也不能跳过，因为 a 的剩余长度(为 1)小于 mState 的绝对值(为 4)。

算法 3 SMCH 中的 Scan 和 Skip 过程

```
1:   procedure Scan(LZ77DATA[0…n-1])
2:     Initialize window of length of max_pat_len
3:     w ← max_pat_len
4:     sw ← 0
5:     for i ← 0 to n-1 do
6:       if LZ77DATA_i is a char then
7:       window_w ← LZ77DATA_i
8:       mState_window_w ← 0
9:       Skip(window, mState_window, sw)
10:      update w and sw
11:    else
12:      define d is distance of LZ77DATA_i, l is length of LZ77DATA_i
13:      copy window_d:d+l-1 to window_w:w+l-1
14:      for j ← 0 to l-1 do
15:        if l-j ≥ abs(mState_window_w:w+l-1) then
16:          mState_window_w+j ← mState_window_d+j
17:        else
18:          mState_windoww+j ← 0
19:        end if
20:        Skip(window, mState_window, sw)
21:      end for
22:      update w and sw
23       end if
24:    end for
25:  end procedure
```

算法 3 显示了 SMCH 中的扫描和跳转的过程。window 是字符的缓冲窗口，w 是字符索引，sw 是搜索索引，w 和 sw 同步但不在同一位置，sw 在 w 之后，mState_window 是 mState 的缓冲窗口，我们支持 w 和 sw 可以从 0 到 window 长度的自动循环。由于模式匹配算法需要一定长度的子串进行搜索，因此算法的第 2~4 行需要根据 window 中的 LZ77 数据初始化一个长度为 max_pat_len 字符串。此时，索引 w 位

于 max_pat_len 位置，搜索索引 sw 位于 0 位置。模式搜索全部在 sw。第 5~24 行是算法 3 的主体，它循环处理 LZ77 数据。第 6~11 行是处理 ASCII 字符的情况，在第 10 行更新缓存的索引。开始时，位于 w 位置的 mState 窗口没有得到匹配的状态，所以在第 8 行设置 $mState\_window_w$ 为 0。第 9 行是搜索部分，它从搜索索引 sw 开始，而不是 w。第 11~23 行是处理指针(length, distance)的过程。第 13 行根据指针(length, distance)更新 window。第 14~22 行是跳过的关键，通过扫描复制字符串，如果在 w 位置处的已知指针长度大于等于 mState 的绝对值，则可以确定该位置的 mState 值可以重复使用指针指向相应位置处的 mState 值，否则设置为 0。该算法在第 10 行和第 22 行有 sw 和 w 对来更新操作。第 8 行和第 15~19 行是在位置 w 处 mState 的过滤过程，只有通过该位置，sw 到达时，mState 才有效。

算法 4 算法 3 中的 Search 函数

```
1:    procedure Search(window, mState_window, sw)
2:      if mState_windowsw > 0 then
3:          There is a matching, skip searching here
4:      else if mState_windowsw < 0 then
5:          There is no matching, skip searching here.
6:      else
7:          mState_windowsw ← Search(windowsw:sw+max_len_pat-1)
8:      end if
9:    end procedure
```

在算法 3 中第 9 行和第 20 行，Skip()函数用于搜索一个字符段，它根据 mState 的值确定是否需要模式搜索。Skip()的过程如算法 4 所示。如果 mState 大于 0，则则从该位置开始向后有一个匹配的子字符串。如果 mState 小于 0，则从该位置后向没有成功匹配的子字符串。以上两种情况都可以跳过。但如果 mState 等于 0，则无法判断匹配信息，所以直接用 Search()进行搜索(见 3.2 节)。

根据算法 3 和算法 4，我们可以看到 mState 主要由 sw 处的 Search()函数更新，但算法 3 中 w 处的 mState 信息需要在第 8 行和 15~19 行进行过滤。mState 只有在剩余拷贝长度 l-i 大于或等于 mState 绝对值(算法 3 第 15 行)时才有效，然后可以直接使用搜索指针来使用 mState 信息。

4. 试验评估

4.1. 实验设计

本文的实验规则集为 Snort [20]、ModSecurity [21]和 random rules，Snort 和 ModSecurity 的特点如表 2 所示。目标数据集是从 Yahoo！网站爬取的 50 MB 网页数据。实验环境是 g++4.8.4，Ubuntu 14.04，2 核 intel e5-2620 v3 CPU，2 GB 内存。

Table 2. The characteristics of rule set
表 2. 规则集的特征

Rule Set	Number of Patterns	Avg. Length	Max Length	Min Length
Snort	5029	16.1	122	2
ModSecurity	3344	18.2	84	2

SMCH 算法在不改变原算法数据结构的前提下，跳过了一些字符，因此我们主要考虑跳过率和加速率。跳过率是跳过的字符数与正常文本长度的比。加速比是解压缩和匹配的总时间(decompress & match)与压缩匹配(Compressed Match)的总时间之比。前者包括 GZIP 数据解压为纯文本数据的时间和原始模式匹配算法在纯文本上搜索匹配的时间。后者包括 GZIP 数据被解压成 LZ77 数据的时间和 SMCH 算法处理 LZ77 数据的时间。实验输出全部归一化。

$$\text{skip ratio} = \frac{\text{number of skipped characters}}{\text{length of plain text}} \times 100\% \quad (1)$$

$$\text{accelerate ratio} = \frac{\text{total time of decompress \& match}}{\text{total time of compressed match}} \times 100\% \quad (2)$$

值得一提的是，与匹配的时间相比，解压缩的时间非常少，并且 GZIP 数据解压为 LZ77 数据的时间比将 GZIP 数据解压为原始文本数据的时间要小一些。因此，我们在总时间里把它们放在一起讨论。

4.2. 性能评估

在本节中，我们主要讨论 SMCH 算法的性能。实验中使用了公共规则集(Snort 和 ModSecurity)。实验结果如图 7 所示，可以看出，SMCH 算法可以大大提高原模式算法的匹配速度。与原始 WM 算法相比，SMCH 在 Snort 和 ModSecurity 中的速度分别提高了 441%和 385%。

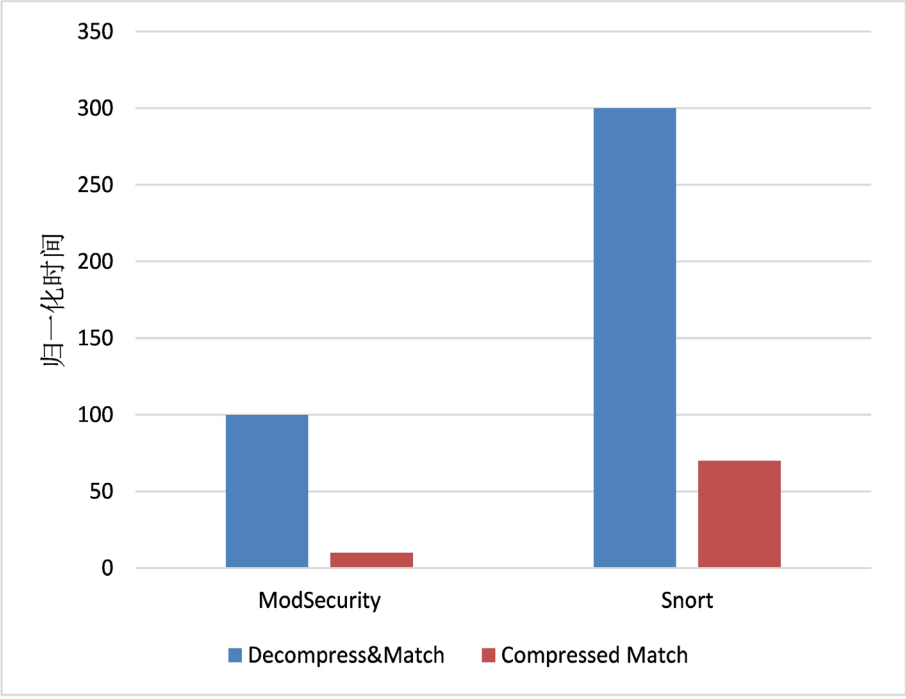

Figure 7. Normalize time
图 7. 归一化时间

此外，本文还选择了两种典型算法 ACCH [16]和 SPC [17]作为对比。图 8(a)显示了算法的加速性能。与原有的模式匹配算法相比，SMCH 算法的性能有了很大的提高。SMCH 算法在 Snort 和 ModSecurity 中的加速比分别为 5.41 和 4.85。另外，无论是 ModSecurity 还是 Snort，SMCH 的加速比都大于 ACCH 和 SPC，它们都小于 2.00。

如图 8(b)所示，SMCH 算法在 ModSecurity 中的跳转率可以达到 91.6%左右，在 Snort 中可以达到 86.8%左右。在 ModSecurity 中，SMCH 的跳过率略高于 ACCH 和 SPC；在 Snort 中，SMCH 的跳转率远远高于 ACCH 和 SPC，SMCH 的跳转条件非常简单，因此 SMCH 可以比 ACCH 和 SPC 跳过更多的字符。

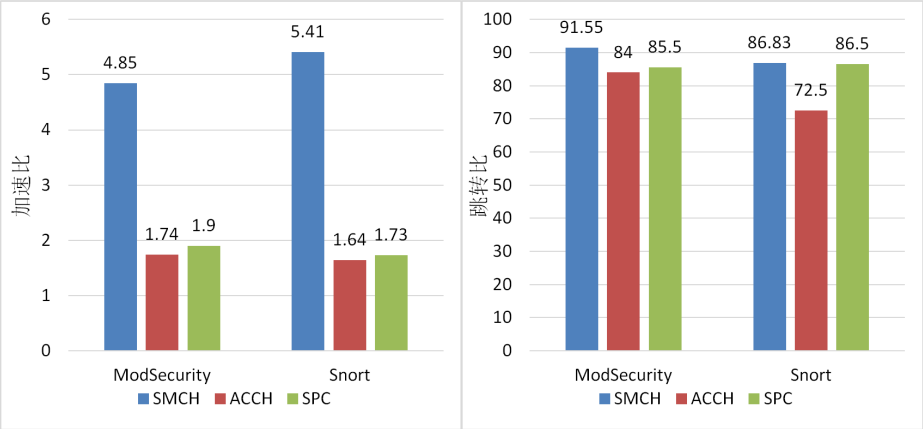

Figure 8. Accelerate rate & skip rate
图 8. 加速率和跳转率

4.3. 算法的扩展性

在本节中，我们将探讨 SMCH 算法的可伸缩性。主要考虑两个因素：模式的平均长度和模式的总数。由于 SMCH 算法中的跳转与模式的长度有关，而 WM 算法在大规模模式应用场景中仍然能够保持稳定的性能。因此，我们基于这两个因素测试算法的可扩展性。实验中，以随机方式生成模式集，目标文本不变。实验结果如图 9 和图 10 所示。

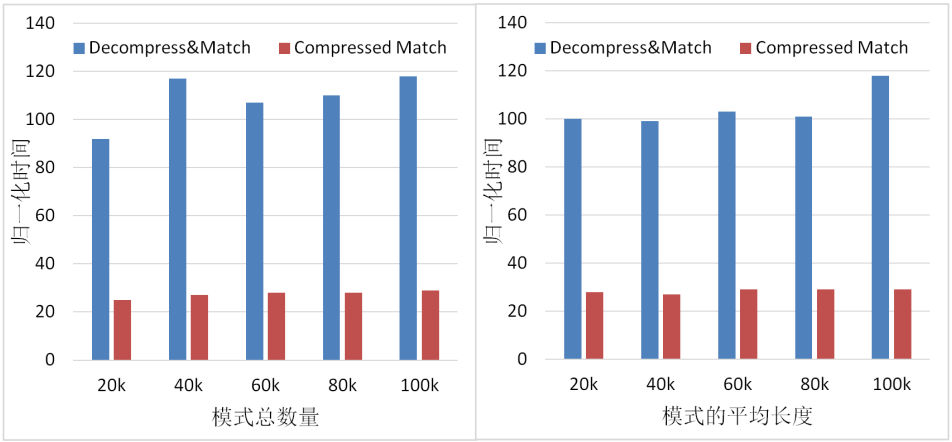

Figure 9. Normalize time when k = 1000
图 9. k = 1000 时的归一化时间

在图 9(a)中，模式的总数从 20,000 增加到 100,000，并且模式的平均长度为 10。在图 9(b)中，模式的平均长度在 5 到 30 之间，模式的总数为 20,000。可以看出，SMCH 算法的速度基本稳定。同时，随机规则中 SMCH 的加速比也很高。显然，SMCH 算法不会破坏原有的算法结构，因此 SMCH 算法具有很好的可扩展性。

图 10(a)和图 10(b)是 SMCH 算法的跳过率的结果。结果表明，模式总数和模式平均长度对跳过率影响不大。SMCH 算法中的跳转与模式长度有关。模式长度越长，跳越的可能性越小。但如果模式很长，匹配的可能性也很小。另外，如果匹配失败，仍然可以返回 mState。综上所述，跳转虽然与模式长度有关，但跳转率不受影响。

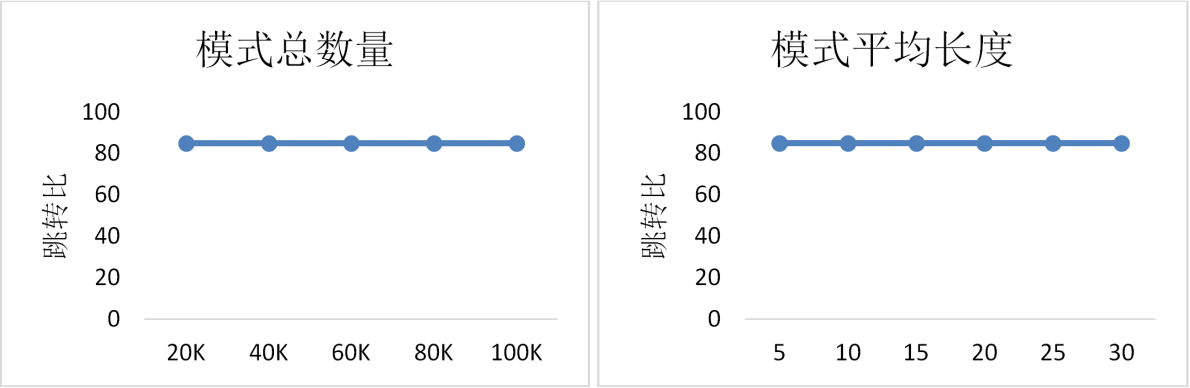

Figure 10. Jump rate when k = 1000
图 10. k = 1000 时的跳跃率

SMCH 算法具有较高的跳过率、高加速比和可扩展性。因此，与原有的模式匹配算法相比，SMCH 算法在不增加其他负担的情况下，可以很好地与原算法共存，甚至可以获得非常高的性能提升。

5. 总结

HTTP 压缩技术已经得到了广泛的应用，因此我们应该利用压缩数据的特征，来提高匹配性能。本文提出的 SMCH 算法可以充分利用压缩数据的特性，提高原模式算法的性能。mState 作为 SMCH 与模式匹配算法之间的一个纽带，使得 SMCH 对模式匹配算法的依赖性降低，在不改变原算法结构的情况下，可以与模式匹配算法无缝结合，从而获得 SMCH 的强大应用价值。

参考文献

[1] Keyword Research, Competitive Analysis, Website Ranking, Alexa. http://www.alexa.com

[2] Usage Statistics of Compression for Websites. https://w3techs.com/technologies/details/ce-compression/all/all

[3] The GZIP Home Page. http://www.gzip.org

[4] Ziv, J. and Lempel, A. (1977) A Universal Algorithm for Sequential Data Compression. *IEEE Transactions on Information Theory*, **23**, 337-343. https://doi.org/10.1109/TIT.1977.1055714

[5] Huffman, D. (1952) A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of IRE*, **40**, 1098-1101. https://doi.org/10.1109/JRPROC.1952.273898

[6] Knuth, D.E., Morris, J.H. and Pratt, V.R. (1977) Fast Pattern Matching in Strings. *SIAM Journal on Computing*, **6**, 323-350. https://doi.org/10.1137/0206024

[7] Aho, A. and Corasick, M. (1975) Efficient String Matching: An Aid to Biblio-Graphic Search. *Communications of the ACM*, **18**, 333-340. https://doi.org/10.1145/360825.360855

[8] Karp, R.M. and Rabin, M.O. (1987) Efficient Randomized Pattern-Matching Algorithms. *IBM Journal of Research and Development*, **31**, 249-260. https://doi.org/10.1147/rd.312.0249

[9] Wang, Q., Lu, Y.H., Liu, Y., Liu, Y.B., Tan, J.L. and Sun, B. (2017) Dynamical Adaptive Karp-Rabin Pattern Matching Algorithm. *CEA*, **53**, 39-44.

[10] Wu, S. and Manber, U. (1994) A Fast Algorithm for Pattern Searching.

[11] Boyer, R. and Moore, J. (1977) A Fast String Searching Algorithm. *Communications of the ACM*, **20**, 762-772. https://doi.org/10.1145/359842.359859

[12] Cantone, D., Faro, S. and Giaquinta, E. (2012) Adapting Boyer-Moore-Like Algorithms for Searching Huffman Encoded Texts. *International Journal of Foundations of Computer Science*, **23**, 343-356. https://doi.org/10.1142/S0129054112400163

[13] Bremler-Barr, A., David, S.T., Hay, D., *et al.* (2012) Decompression-Free Inspection: DPI for Shared Dictionary Compression over HTTP. 2012 *Proceedings IEEE*, Orlando, 25-30 March 2012, 1987-1995. https://doi.org/10.1109/INFCOM.2012.6195576

[14] Navarro, G. and Tarhio, J. (2000) Boyer Moore, String Matching over Ziv-Lempel Compressed Text. In: *Annual Symposium on Combinatorial Pattern Matching*, Springer, Berlin Heidelberg, 166-180. https://doi.org/10.1007/3-540-45123-4_16

[15] Gawrychowski, P. (2013) Optimal Pattern Matching in LZW Compressed Strings. *ACM Transactions on Algorithms* (*TALG*), **9**, 25. https://doi.org/10.1145/2483699.2483705

[16] Bremler-Barr, A. and Koral, Y. (2009) Accelerating Pattern Matching on Compressed HTTP Traffic. 2009 *Proceeding IEEE INFOCOM*, Rio de Janeiro, 19-25 April 2009, 397-405. https://doi.org/10.1109/INFCOM.2009.5061944

[17] Bremler-Barr, A., Koral, Y. and Zigdon, V. (2011) Shift-Based Pattern Matching for Compressed Web Traffic. *IEEE* 12*th International Conference on High Performance Switching and Routing*, Cartagena, 4-6 July 2011, 222-229. https://doi.org/10.1109/HPSR.2011.5986030

[18] Bremler-Barr, A. and Koral, Y. (2012) Accelerating Multipattern Matching on Compressed HTTP Traffic. *IEEE*/*ACM Transactions on Networking*, **20**, 970-983. https://doi.org/10.1109/TNET.2011.2172456

[19] Hassan, M. (2016) Fast Pattern Matching Algorithm on Compressed Network Traffic. *China Communications*, **13**, 141-150. https://doi.org/10.1109/CC.2016.7489982

[20] Snort-Network Intrusion Detection, Prevention System. https://www.snort.org

[21] ModSecurity: Open Source Web Application Firewall. https://modsecurity.org