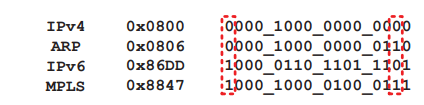
**HyperParser解析器**

# 目的

HyperParser解析器是专用于HyperParser的解析器，其目的在于对指定格式的输入文件进行解析，获取其所表示的所有可能数据包头，并对其进行压缩，去除重复信息，解出关键bit，从而利用关键bit对数据包进行识别。



如图，对于图中的协议类型字段，只需要识别2bit就可以辨别出其协议类型，而无须检查所有16bit。

需要说明的是，本解析器仅针对预设网络生效，如网络中出现未知数据帧，则无法正确解析。例如，01解析为A，10解析为B，出现00或11则解析为未知。

# 文件目录

code：存放解析器代码。

doc：存放文档，包括本说明文档和一篇论文。

source：存放输入文件，即预设网络帧格式。

txt：存放所有输出文件，包括临时中间结果和最终结果。

# 输入文件

本部分以source文件夹下headers-simple.txt为例对输入文件格式进行说明。

对于某一特定数据帧而言，其帧头由多个域组成。在headers-simple.txt中，指定了在该网络中仅存在ethernet域、ieee802-1q域、ipv4域、tcp域、udp域，对于一个特定的数据帧，这些域不一定同时出现。在一个域中，有fields和next\_header，其中fields表示该域的数据结构，next\_header表明这些域的相互关系。

ethernet {

fields {

dstAddr : 48 : extract,

srcAddr : 48 : extract,

etherType : 16 : extract,

}

next\_header = map(etherType) {

0x8100, 0x9100, 0x9200, 0x9300 : ieee802-1q,

0x0800 : ipv4,

}

# This is some test code

}

例如，对于ethernet域而言，其数据结构由48bit的dstAddr、48bit的srcAddr和16bit的etherType组成。当etherType值为0x8100, 0x9100, 0x9200, 0x9300中的任意一个时，下一个域为ieee802-1q；当etherType值为0x0800时，下一个域为ipv4。对于一个符合约定的网络数据帧而言，etherType值不应出现约定值之外的数值。如果出现，有可能造成HyperParser的解析错误，或分类为未知。

ieee802-1q {

fields {

pcp : 3 : extract,

cfi : 1,

vid : 12 : extract,

etherType : 16 : extract,

}

next\_header = map(etherType) {

0x8100, 0x9100, 0x9200, 0x9300 : ieee802-1q,

0x0800 : ipv4,

}

max\_var = vlan

max = 2

}

对于ieee802-1q而言，由于下一个域有可能指向它自己，因此，需要指定连续出现ieee802-1q域的最大重复次数，这个值由max参数指定。因此，在本网络中，最多出现连续两个ieee802-1q域。

ipv4 {

fields {

version : 4,

ihl : 4,

diffserv : 8,

totalLen : 16,

identificaiton : 16,

flags : 3,

fragOffset : 13,

ttl : 8,

protocol : 8,

hdrChecksum : 16,

srcAddr : 32,

dstAddr : 32,

options : \*,

}

next\_header = map(fragOffset, protocol) {

6 : tcp,

17 : udp,

}

length = ihl \* 4 \* 8

max\_length = 256

}

在ipv4域中，fields中的options字段为可选字段，\*表示其长度不定，具体长度取值由length和max\_length共同指定。length表示其取值，为ihl\*4\*8，即ihl\*32。由于在fields中ihl为4bit长度，因此ihl值可能为0到15之间的一个整数。max\_length为整个ipv4域的长度，减去前面确定字段的长度，得到options最大长度为96。因此，options的长度可能取值为0\*32、1\*32、2\*32、3\*32。

tcp {

fields {

srcPort : 16,

dstPort : 16,

seqNo : 32,

ackNo : 32,

dataOffset : 4,

res : 3,

ecn : 3,

ctrl : 6,

window : 16,

checksum : 16,

urgentPtr : 16,

options : \*,

}

length = dataOffset \* 4 \* 8

max\_length = 192

}

同理，对于tcp域，options的长度可能取值为0\*32、1\*32。由于tcp域没有next\_header，因此在每次解析中，其为最后一个可能的域。

在headers-service\_provider-fixed.txt中出现的mpls域本解析器无法解析。

mpls {

fields {

label : 20 : extract,

tc : 3 : extract,

bos : 1 : extract,

ttl : 8 : extract,

}

pseudo-fields {

next-header : 4,

}

next\_header = map(bos, next-header) {

b0xxxx : mpls,

b10100 : ipv4,

b10110 : ipv6,

}

next\_header\_def = b10001

max = 5

length = 32

}

该域可以转化为headers-service\_provider-prog.txt中的mpls域和mpls-pseudo域，此时就可以解析了。

mpls {

fields {

label : 20 : extract,

tc : 3 : extract,

bos : 1 : extract,

ttl : 8 : extract,

}

next\_header = map(bos) {

b0 : mpls,

b1 : mpls-pseudo,

}

next\_header\_def = b1

max = 5

length = 32

}

mpls-pseudo {

pseudo-fields {

next-header : 4,

}

next\_header = map(next-header) {

b0100 : ipv4,

b0110 : ipv6,

}

next\_header\_def = b0001

}

需要注意的是，mpls-pseudo并不是一个实际存在于数据包头的域，其下一个域只能是ipv4或ipv6。而在ipv4和ipv6中，第一个字段即为4bit的version。mpls-pseudo可以理解为下一个ipv4或ipv6域中“独立”出来的version字段。因此，本解析器在进行处理时，mpls-pseudo的下一个域会相应删去前4bit。

# 处理流程

处理共分为三个阶段。

1. 阶段一，文件解析

阶段一首先对输入的文件进行解析，解析的输入文件是source目录下的某一网络数据帧头格式文件，输出文件存放于txt目录，如下：

bitmap\_out.txt：各域的bit位串，其中1表示该bit位参与next\_header的判定，0表示不参与。

graph\_out.txt：域间转换关系，罗列出所有可能出现的域转换路径。

next\_header\_out.txt：域转换方式，以指定格式表达该域参与next\_header判定的bit在何值下会跳转到何域。该文件不表达参与next\_header判定的bit位置信息，配合bitmap\_out.txt文件使用。

option\_out.txt：罗列各域option的可能长度（单位：bit）。

output.txt：将所有可能的域转换路径转换为相应的bit位串，用1表示参与next\_header判断的bit位，0表示不参与。

以“headers-simple.txt”为例，共有5个域：ethernet、ieee802-1q、ipv4、tcp、udp。

——bitmap\_out.txt——

ethernet

0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000001111111111111111

ieee802-1q

00000000000000001111111111111111

……

对于ethernet域而言，其数据结构由48bit的dstAddr、48bit的srcAddr和16bit的etherType组成，仅有etherType参与next\_header的判定，因此在bitmap\_out.txt中，表现为96bit的0和16bit 的1。

——graph\_out.txt——

ethernet->ieee802-1q->ieee802-1q(1)->ipv4->tcp

ethernet->ieee802-1q->ieee802-1q(1)->ipv4->udp

ethernet->ieee802-1q->ipv4->tcp

ethernet->ieee802-1q->ipv4->udp

ethernet->ipv4->tcp

ethernet->ipv4->udp

依据域间的跳转关系，构建一张图，通过对图的遍历，就可以获取域间转换关系所有的可能结果。需要注意的是，在具体实现时，需要添加一个空节点作为遍历的起始点和结束点。

——next\_header\_out.txt——

ethernet : (0x8100, 0x9100, 0x9200, 0x9300)->ieee802-1q, (0x0800)->ipv4,

ieee802-1q : (0x8100, 0x9100, 0x9200, 0x9300)->ieee802-1q, (0x0800)->ipv4,

ieee802-1q(1) : (0x8100, 0x9100, 0x9200, 0x9300)->ieee802-1q, (0x0800)->ipv4,

ipv4 : (6)->tcp, (17)->udp,

tcp :

udp :

“next\_header\_out.txt”以另一种结构将分析结果进行存储，主要描述“bitmap\_out.txt”文件中，标记为1的bit位置取何值时，下一个域对应为何域。

——option\_out.txt——

ethernet : 0,

ieee802-1q : 0,

ieee802-1q(1) : 0,

ipv4 : 0, 32, 64, 96,

tcp : 0, 32,

udp : 0,

“option\_out.txt”将option字段的可能取值进行记录。在本解析器中，option选项是可选的。当关闭时，认定option为0；否则，在生成最终结果时，需要将不同option取值考虑进去。例如，tcp的option有0和32，因此tcp字段可能有两种长度的bit位串。

“output.txt”文件实际上是将“graph\_out.txt”中的各个域替换为“bitmap\_out.txt”中相应的bit位串。在替换时，需要考虑“option\_out.txt”中的option可选值。例如，tcp的option有0和32，且解析器选择需要考虑option，那么有关tcp的域转换路径需要进行翻倍，一种是tcp替换为option为0长的，一种是tcp替换为option是32长的。额外长度的option字段，对应bit位均填充为0。上述示例文件均是在关闭option开关时产生的，即认定option为0。

1. 阶段二，填充

经过第一个阶段，获得了特定网络下合法数据帧头bit位串的所有可能模式。在模式中，1表示该bit有可能参与判断，0表示该bit不参与判断。这里，说1表示的bit位是可能参与判断的原因是，如果可能值均相同，那么其本身其实并不参与判断。例如，ethernet域中，为1的bit位置取值为0x8100, 0x9100, 0x9200, 0x9300时，下一个域为ieee802-1q；为1的bit位置取值为0x0800时，下一个域为ipv4。但实际上，由于为1的bit位置低8bit总是0x00，因此并不真正参与判定。因此，阶段二需要将各个判定值插入这些为1的bit位置中去。同时，原先bit位置为0的值，表示不参与判定，因此用x进行替代，表示这个bit位置可能是0也可能是1。

注意到由于多个值可能映射到相同的一个域，因此经过阶段二，有可能出现类似option的路径拓展。例如，对于关闭option选项的“headers-simple.txt”，阶段二之前的“output.txt”文件有6行，而经过阶段二生成的“setone\_output.txt”有42行。

阶段二生成“setone\_graph\_out.txt”和“setone\_output.txt”。其中，“setone\_output.txt”表示在该指定网络中，所有可能出现的合法数据帧头格式，x表示该bit位可能是0也可能是1。“setone\_graph\_out.txt”表示“setone\_output.txt”中相应行数的域转换关系。

1. 阶段三，提取

经过阶段二，已经成功获取所有可能出现的合法数据帧头格式，阶段三就是对他们进行提取，找到关键bit位，使得仅依赖关键bit位的值，就能区分该数据帧头对应哪一种域转换路径。

——预备处理部分——

步骤一：将数据帧头进行补齐，不够长的数据帧头均填充x，使得所有数据帧头一样长。

步骤二：检查所有列，若某列bit位均为x，则删除该列，同时记录所保留列编号。

步骤三：删除重复列，若某列和前面出现的某列完全一致，则删除该列。记录所保留列编号。

步骤四：检查所有列，若某列bit位仅出现x和1，或仅出现x和0，则删除该列。记录所保留列编号。

步骤五：检验保留的列能否区分所有行。方法是，遍历所有可能的两两行组合，检验其是否匹配。若均不匹配，则认为可以区分所有行，保留步骤四的结果。若出现匹配的，则保留步骤三的结果。

——————————

在预备处理部分中，步骤一到步骤三无疑是没有信息量的损失的。对于步骤四，这里举例进行说明。

设位串长度为1，假设为：

位串A：1

位串B：x

当送入一个位串“0”时，可以判定符合位串B，因为不可能是位串A；但当送入一个位串“1”时，就无法进行分辨了。这种情况下，其所能提供的信息量和输入相关。现在假设有两个这种位串，检验其能否相互补充。

设位串长度为2，假设为：

位串C：0x

位串D：x1

当送入“00”、“11”时，可以进行判定；当送入“10”时，认定是非法输入；当送入“01”时，无法进行判定。

由于某列bit位仅出现x和1，或仅出现x和0时，该bit位所能提供的信息量无法保证，因此在预备处理部分将这些bit进行删除。

经过预备处理部分，剩下的bit位串长度已经相对较小了，但并不是最小的。要继续进行缩小，典型的办法是进行暴力搜索。

­­——进一步处理部分（暴力搜索）——

设位串长度为，需要分类的域转换路径共有个。此时，最好情况下，仅需要 bit就可以区分这种域转换路径。因此，暴力搜索的初始列数量设置为=。

步骤一：设置选择列数量为初始列数量。

步骤二：获取从列中选择列的所有可能组合。遍历这些组合，直到遇到所选择的列能够区分所有行，进入步骤四。如果所有组合均不能，进入步骤三。

步骤三：设置值自增1，返回步骤二。

步骤四：记录所选择的列，返回选择结果。

—————————————————

但在实际操作中，这种暴力搜索所带来的时间复杂度是巨大的，原因就在于庞大的计算量。因此，产生了局部最优的办法。

——进一步处理部分（局部最优）——

设位串长度为，需要分类的域转换路径共有个。

步骤一：设置。

步骤二：从列中任选一列，判定删除该列后剩下的列能否区分所有行。若能，进入步骤四；否则，进入步骤三。

步骤三：重新从列中任选一列，但不能与当前值下已选择过的列重复。判定删除该列后剩下的列能否区分所有行。若能，进入步骤四；否则，进入步骤三。若列中的所有列均已尝试过无法再进行删除，进入步骤五。

步骤四：设置自减1，将位串更新为删减后的位串。记录保留信息。进入步骤二。

步骤五：记录剩余所选择的列编号，返回选择结果。

—————————————————

局部最优的进一步处理方式减小了非常大的计算量，可以在能否接受的时间范围内返回结果。其代价是无法保证输出结果一定是最优的。

阶段三的最终输出结果文件是x10select\_further2\_info.txt和x10select\_further2\_output.txt。

# 代码处理

在code目录下，打开main.py文件。需要设置的参数包括：file文件的选择、option\_sw的开关、space\_num的设置。其中，file文件选择的是具体输入哪个预设网络；option\_sw确定是否打开option选项；space\_num用于控制阶段一产生的结果文件“output.txt”中，各域之间的空格数量，便于查看。

下面给出各个预设网络在关闭option情况下运行的典型时间。

* headers-big\_union.txt：大于10分钟。由于没有实际跑完，不确定实际耗时
* headers-datacenter.txt：22.195秒
* headers-edge.txt：0.187秒
* headers-enterprise.txt：14.110秒
* headers-service\_provider-fixed.txt：执行错误（fixed格式需要转换为prog）
* headers-service\_provider-prog.txt：0.050秒
* headers-simple.txt：0.082秒

# 输出文件

在最终输出的文件x10select\_further2\_info.txt和x10select\_further2\_output.txt中，分别记录了所选择的列下标以及选择的结果。这些所选择的列下标信息，就是需要的关键bit位置。