NDPI* - Deep Packet Inspection Source Code Analyse

ZhangYang

December 12, 2022

Contents

1 Introduction

nDPI is a network protocol analyzer, which extended the openDPI libs. It can extract application layer protocol for a given packet, such as wechat, qq, youtube, etc. Further more, customize protocols are also supported and be defined and added to nDPI.

nDPI allows application-layer detection of protocols, regardless of the port being used. This means that it is possible to both detect known protocols on non-standard ports (e.g. detect HTTP on ports other than 80), and also the opposite (e.g. detect Skype traffic on port 80). This is because nowadays the relationship between port and the application no longer holds.

1.1 Download Source

nDPI source code can be downloaded from `https://github.com/ntop/nDPI`. A ndpi-dev package is also available from `http://packages.ntop.org/` including binary libraries and header files.

1.2 Compiling nDPI Source Code

Start using nDPI library is very simple. In order to compile this library you must meet certain prerequisites such as:

GNU autotools/libtool

gawk

gcc

To do this you need to install them using the following commands:

Fedora: yum groupinstall Development tools; yum install automake libpcap-devel gcc-c++ libtool

Debian: apt-get install build-essential libpcap-dev

Mac OSX: port install XXX (Please install macports)

Once done that, you can compile the nDPI source code as follows:

```
cd <nDPI source code directory>

./autogen.sh

./configure

make
```

After compiling, nDPI will generate two components: one is the **xt_ndpi.ko**, which can be added to linux kernel as a module; the other is the lib, used by ndpiReader, which is an application example in nDPI.

1.3 Compiling the demo ndpiReader Source Code

Starting using ndpiReader demo is also simple. In order to compile this you must use the following command:

```
cd <nDPI source code directory>/example

make
```

Where this paragraph is directly copy from **"nDPI - Quick Start Guide"**. In practice, we found the ndpiReader is generated after type make for compiling the nDPI source code.

2 ndpiReader

ndpiReader is an example tool which uses ndpi lib. ndpiReader is able to read from a pcap file or catpure traffic from a network interface and process it with ndpi lib. It implements only some basic features just to show what can be done with ndpi libs.

2.1 ndpiReader Command Line Options

The demo **ndpiReader** application can be used both in terms of speed/features analysis and encapsulations support. In particular, it is possible to specify a lot of command line options.

After typing **"ndpiReader -h"**, the available options and a minimal explanation of each option are listed below:

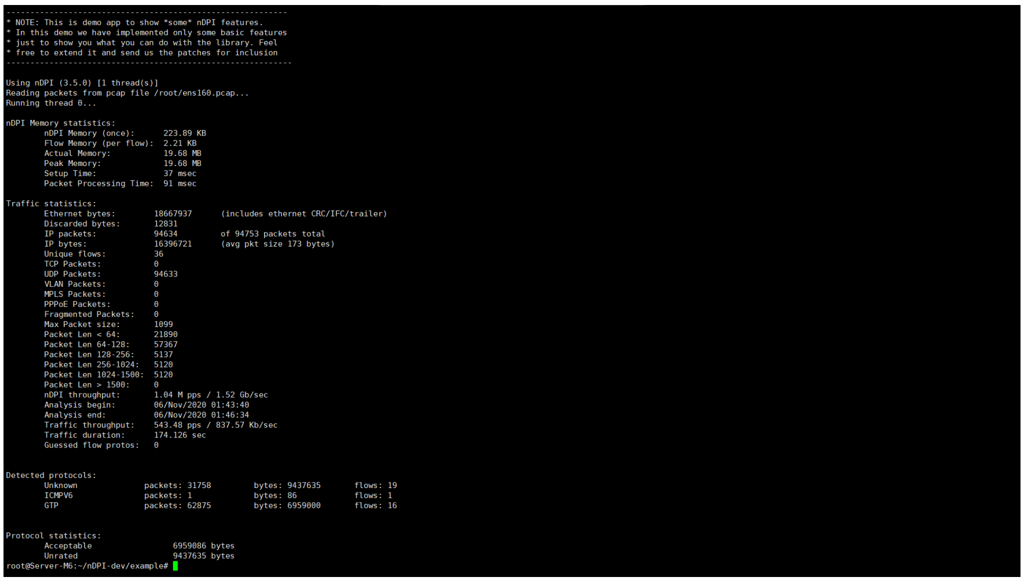
```
ndpiReader -i <file | device> [-f <filter>][-s <duration>][-m <duration>][-b <num bin clusters>]
    [-p <protos>][-l <loops> [-q][-d][-J][-h][-D][-e <len>][-t][-v <level>]
    [-n <threads>][-w <file>][-c <file>][-C <file>][-j <file>][-x <file>]
    [-T <num>][-U <num>] [-x <domain>]
```

The details of these options are shown as:

```
Usage:
  -i    <file | device>        | Specify a pcap file/playlist to read packets from or a
                               | device for live capture (comma-separated list)
  -f    <BPF filter>           | Specify a BPF filter for filtering selected traffic
  -s    <duration>             | Maximum capture duration in seconds (live traffic capture only)
  -m    <duration>             | Split analysis duration in ¡duration¿ max seconds
  -p    <file>.protos          | Specify a protocol file (eg. protos.txt)
  -l    <num loops>            | Number of detection loops (test only)
  -n    <num threads>          | Number of threads. Default: number of interfaces in -i.
                               | Ignored with pcap files.
  -b    <num bin clusters>     | Number of bin clusters
  -g    <id:id...>             | Thread affinity mask (one core id per thread)
  -d                           | Disable protocol guess and use only DPI
  -e    <len>                  | Min human readeable string match len. Default 5
  -q                           | Quiet mode
  -J                           | Display flow SPLT (sequence of packet length and time)
                               | and BD (byte distribution). See https://github.com/cisco/joy
  -t                           | Dissect GTP/TZSP tunnels
  -P    <a>:<b>:<c>:<d>:<e>    | Enable payload analysis:
                               | <a> = min pattern len to search
                               | <b> = max pattern len to search
                               | <c> = max num packets per flow
                               | <d> = max packet payload dissection
                               | <d> = max num reported payloads
                               | Default: 4:8:10:128:25
  -r                           | Print nDPI version and git revision
  -c    <path>                 | Load custom categories from the specified file
  -C    <path>                 | Write output in CSV format on the specified file
  -w    <path>                 | Write test output on the specified file. This is useful for
                               | testing purposes in order to compare results across runs
  -h                           | This help
  -v    <1|2|3>                | Verbose 'unknown protocol' packet print.
                               | 1 = verbose
                               | 2 = very verbose
                               | 3 = port stats
  -V    <1-4>                  | nDPI logging level
                               | 1 - trace, 2 - debug, 3 - full debug
                               | >3 - full debug + log enabled for all protocols (i.e. '-u all')
  -u    all|proto|num[,...]    | Enable logging only for such protocol(s)
                               | If this flag is present multiple times (directly, or via '-V'),
                               | only the last instance will be considered
  -T    <num>                  | Max number of TCP processed packets before giving up [default: 80]
  -U    <num>                  | Max number of UDP processed packets before giving up [default: 16]
  -D                           | Enable DoH traffic analysis based on content (no DPI)
  -x    <domain>               | Check domain name [Test only]
```

2.2 ndpiReader Examples

As the options shown, user can simply type **"ndpiReader -i <filename>"** to get the results of a pcap file, as shown in Fig.1.



```
----------------------------------------------------------
* NOTE: This is demo app to show *some* nDPI features.
* In this demo we have implemented only some basic features
* just to show you what you can do with the library. Feel
* free to extend it and send us the patches for inclusion
------------------------------------------------------------

Using nDPI (3.5.0) [1 thread(s)]
Reading packets from pcap file /root/ens160.pcap...
Running thread 0...

nDPI Memory statistics:
        nDPI Memory (once):      223.89 KB
        Flow Memory (per flow):  2.21 KB
        Actual Memory:           19.68 MB
        Peak Memory:             19.68 MB
        Setup Time:              37 msec
        Packet Processing Time:  91 msec

Traffic statistics:
        Ethernet bytes:        18667937      (includes ethernet CRC/IFC/trailer)
        Discarded bytes:       12831
        IP packets:            94634         of 94753 packets total
        IP bytes:              16396721      (avg pkt size 173 bytes)
        Unique flows:          36
        TCP Packets:           0
        UDP Packets:           94633
        VLAN Packets:          0
        MPLS Packets:          0
        PPPoE Packets:         0
        Fragmented Packets:    0
        Max Packet size:       1099
        Packet Len < 64:       21890
        Packet Len 64-128:     57367
        Packet Len 128-256:    5137
        Packet Len 256-1024:   5120
        Packet Len 1024-1500:  5120
        Packet Len > 1500:     0
        nDPI throughput:       1.04 M pps / 1.52 Gb/sec
        Analysis begin:        06/Nov/2020 01:43:40
        Analysis end:          06/Nov/2020 01:46:34
        Traffic throughput:    543.48 pps / 837.57 Kb/sec
        Traffic duration:      174.126 sec
        Guessed flow protos:   0


Detected protocols:
        Unknown              packets: 31758        bytes: 9437635       flows: 19
        ICMPV6               packets: 1            bytes: 86            flows: 1
        GTP                  packets: 62875        bytes: 6959000       flows: 16


Protocol statistics:
        Acceptable                   6959086 bytes
        Unrated                      9437635 bytes
root@Server-M6:~/nDPI-dev/example# 
```

Figure 1: An example of ndpiReader.

Note that the resolving of runtime traffic is also supported in **ndpiReader**.

It can be seen the output of **ndpiReader** contains 4 statistic: the memory statistic, traffic statistic, detected protocols statistic and protocol statistic. Among them, traffic statistic and protocol statistic contains the statistic of L4 and L7 packets, respectively; detected protocols show all the protocols found in the pcap file.

2.3 Source Code Analysis

This section begins to analyze the source code of **ndpiReader**, all the functions belong to the ndpi libs will be analyzed in other sections.

The entry of **ndpiReader** is as below:

```
int main(int argc, char **argv) {
    ...
    ndpi_info_mod = ndpi_init_detection_module(ndpi_no_prefs);

    dgaUnitTest();
    ...

    for(i=0; i<num_loops; i++)
        test_lib();


    ...
}
```

The **main** function contains 3 parts, **ndpi_init_detection_module** is the entry of nDPI initialization which will be introduced in section 4.1; **dgaUnitTest** and later omit parts are the test function, which validates some components in the nDPI; after that, **test_lib** will create multi-threads and begin to analyze traffic, as below:

Listing 1: 1st Part of **test_lib**

```
void test_lib () {
    ...
    for(thread_id = 0; thread_id < num_threads; thread_id++) {
        cap = openPcapFileOrDevice(thread_id,
            (const u_char*) _pcap_file [thread_id]) ;
        setupDetection(thread_id, cap);
    }
```

openPcapFileOrDevice is used to open a pcap file or DPDK device, then each packets in the pcap file or each runtime packet will be collected for later usage.
setupDetection is the initialization entry of **ndpiReader**, which will be introduced in section 2.3.1.

Listing 2: 2nd Part of **test_lib**

```
    for(thread_id = 0; thread_id < num_threads; thread_id++) {
        pthread_create(&ndpi_thread_info[thread_id].pthread, NULL,
            processing_thread, (void *) thread_id);
    }

    for(thread_id = 0; thread_id < num_threads; thread_id++) {
        pthread_join(ndpi_thread_info[thread_id].pthread, &thd_res);
    }

    printResults(processing_time_usec, setup_time_usec);

    ...
}
```

The second part creates **num_threads** threads, which can be specified by **"-n"** option, and the default value is **1**. After that, wait on the **pthread_join** function until all the threads exit. Finally, **printResults** will show all the results, which will be described in section 2.3.3.

2.3.1 Initialization

setupDetetion is the function which initializes the **ndpiReader**, as below:

Listing 3: 1st Part of **setupDetection**

```
static void setupDetection(u_int16_t thread_id, pcap_t * pcap_handle) {
    ...
    ndpi_thread_info[thread_id].workflow = ndpi_workflow_init(&prefs, pcap_handle);
```

ndpi_thread_info records all information within a thread. Among them, the most important one is the **workflow**, includes all the structures needed to detect a protocol, and created inside the **ndpi_workflow_init**, which will be analyze in section 4.1.

Listing 4: 2nd Part of **setupDetection**

```
    ndpi_workflow_set_flow_detected_callback(ndpi_thread_info[thread_id].workflow,
        on_protocol_discovered, (void *)(uintptr_t)thread_id);


    NDPI_BITMASK_SET_ALL(all);
    ndpi_set_protocol_detection_bitmask2(ndpi_thread_info[thread_id].workflow->ndpi_struct, &all);
```

The second part of **setupDetection** function first set the callback function, which is invoked when a new protocol is detected. Then call **ndpi_set_protocol_detection_bitmask2** function to add third party dissector to the **workflow**, which will be introduced in section 4.2. Note that **NDPI_BITMASK_SET_ALL** mask **all** with all 1, means allowing any dissector be added.

Listing 5: 3rd Part of **setupDetection**

```
        if(_protoFilePath != NULL)
            ndpi_load_protocols_file(ndpi_thread_info[thread_id].workflow->ndpi_struct, _protoFilePath);

        if(_customCategoryFilePath)
            ndpi_load_categories_file(ndpi_thread_info[thread_id].workflow->ndpi_struct,
                _customCategoryFilePath);

         ndpi_finalize_initalization(ndpi_thread_info[thread_id].workflow->ndpi_struct);
    }
```

In the last part, if exists, the custom protocol and category file will be load and resolved into the **ndpi_str** of **workflow**. Finally, some ac tries which will be introduced in section 3.3 are finalized in the last function **ndpi_finalize_initalization**.

2.3.2 Process

nDPI supports two methods to produce data, one is for static packet through pcap file, the other handle the runtime packets with the program and driver DPDK. Take DPDK as an example:

```
    void * processing_thread(void *_thread_id) {
        ...

        while(dpdk_run_capture) {
            u_int16_t num = rte_eth_rx_burst(dpdk_port_id, 0, bufs, BURST_SIZE);

            for(i = 0; i < num; i++) {
                ...
                ndpi_process_packet((u_char*)&thread_id, &h, (const u_char *)data);
            }
        }
    }
```

After receiving several packets through DPDK lib function **rte_eth_rx_burst**, **processing_thread** will traversal these packets, and each will be handled by ndpi detection function **ndpi_process_packet**, as below:

```
    static void ndpi_process_packet(u_char *args,
                                    const struct pcap_pkthdr *header,
                                    const u_char *packet) {
        ...
        uint8_t *packet_checked = ndpi_malloc(header->caplen);
        memcpy(packet_checked, packet, header->caplen);
        p = ndpi_workflow_process_packet(ndpi_thread_info[thread_id].workflow, header, packet_checked,
            csv_fp);
        ...
    }
```

In the **ndpi_process_packet** function, for each packet, a copy operation generated from **packet** to **packet_checked**. Then enter function **ndpi_workflow_process_packet** for further detection, shown as:

```
    struct ndpi_proto ndpi_workflow_process_packet(...) {
        ...
        return(packet_processing(workflow, time_ms, vlan_id, tunnel_type, iph, iph6,
                                 ip_offset, header->caplen - ip_offset,
                                header->caplen, header, packet, header->ts,
              csv_fp));
    }
```

The omit part is used to collect some basic information from the current received packet, such as the packet size, vlan id, receive time, etc. **packet_processing** will then be called:

Listing 6: 1st Part of **packet_processing**

```
static struct ndpi_proto packet_processing (...) {
    ...
    flow = get_ndpi_flow_info(workflow, IPVERSION, vlan_id,
                              tunnel_type, iph, NULL,
                               ip_offset , ipsize ,
                              ntohs(iph->tot_len) - (iph->ihl * 4),
                              &tcph, &udph, &sport, &dport,
                              &src, &dst, &proto,
                              &payload, &payload_len, &src_to_dst_direction, when);
```

The first thing in **packet_processing** function is to identify or create a flow structure, the detection result will be record in this flow structure. **get_ndpi_flow_info** will be introduced in section 5.1.

Listing 7: 2nd Part of **packet_processing**

```
    if(flow != NULL) {
        ndpi_flow = flow->ndpi_flow;

        if(enable_payload_analyzer && (payload_len > 0))
            ndpi_payload_analyzer(flow, src_to_dst_direction , payload, payload_len, workflow->stats.
                ip_packet_count);
```

If successfully create or find a flow structure through function **get_ndpi_flow_info** and user specify **"-P"** option, the payload analyzer will be called, as below:

```
void ndpi_payload_analyzer(struct ndpi_flow_info *flow, u_int8_t src_to_dst_direction ,
                           u_int8_t *payload, u_int16_t payload_len, u_int32_t packet_id) {
    u_int16_t i, j;
    u_int16_t scan_len = ndpi_min(max_packet_payload_dissection, payload_len);

    for(i=0; i<scan_len; i++) {
        for(j=min_pattern_len; j <= max_pattern_len; j++) {
            if((i+j) < payload_len) {
                ndpi_analyze_payload(flow, src_to_dst_direction , &payload[i], j, packet_id);
            }
        }
    }
}
```

In this function, both **scan_len**, **min_pattern_len** and **max_pattern_len** are determined by **"-P"** option. **ndpi_analyze_payload** records each payload start from **i**th character and last **j** character into a hash structure. Then the result will be printed in **printFlowsStats** function.

Listing 8: 3rd Part of **packet_processing**

```
        if(flow->first_seen_ms == 0)
            flow->first_seen_ms = time_ms;

        flow->last_seen_ms = time_ms;
```

Then, the **first_seen_ms** may be initialized, and **last_seen_ms** is updated. This two variables are for printing usage in **printFlowsStats** function, and also be checked for security detection.

Listing 9: 4th Part of **packet_processing**

```
            if (!flow->has_human_readeable_strings) {
                ...
            }
```

has_human_readeable_strings relates to the data exfiltration detection [2]. The increased monitoring of the protocols has forced attackers to come up with ingenious ways of data exfiltration. One such technique used very successfully in recent years is exfiltration over DNS queries.

The detection procedure match sensitive words by looking into the **bigrams_automa** ac trie, which constructed in **init_string_based_protocols** function, base on the **ndpi_en_bigrams** array, we omit this part of analysis.

Listing 10: 5th Part of **packet_processing**

```
        if (!flow->detection_completed) {
            u_int enough_packets =
                (((proto == IPPROTO_UDP) && ((flow->src2dst_packets + flow->dst2src_packets) >
                    max_num_udp_dissected_pkts))
                || ((proto == IPPROTO_TCP) && ((flow->src2dst_packets + flow->dst2src_packets) >
                    max_num_tcp_dissected_pkts))) ? 1 : 0;

            flow->detected_protocol = ndpi_detection_process_packet(workflow->ndpi_struct, ndpi_flow,
                                                                    iph ? (uint8_t *)iph : (uint8_t *)iph6,
                                                                    ipsize, time_ms, src, dst);
```

Flags **detection_completed** indicates whether this flow need to be detected continuously.

enough_packets is calculated base on the bi-directional total packets number. Note that **enough_packets** only apply to TCP or UDP protocol, for other protocols, as will seen in later part, **detection_completed** is always zero.

After set this two variables, the main detection function, **ndpi_detection_process_packet** will be invoked, which guess the protocol and other information, and will be introduced in section 5.2.

Listing 11: 5th Part of **packet_processing**

```
            if(enough_packets || (flow->detected_protocol.app_protocol !=
                NDPI_PROTOCOL_UNKNOWN)) {
                if ((!enough_packets) && ndpi_extra_dissection_possible(workflow->ndpi_struct, ndpi_flow));
                else {
                    flow->detection_completed = 1;
                    if(flow->detected_protocol.app_protocol == NDPI_PROTOCOL_UNKNOWN) {
                        u_int8_t proto_guessed;
                        flow->detected_protocol = ndpi_detection_giveup(workflow->ndpi_struct, flow->
                            ndpi_flow,
                            enable_protocol_guess, &proto_guessed);
                    }
                }
            }
        }

        return(flow->detected_protocol);
    }
```

For the last part of **packet_processing**, **ndpi_extra_dissection_possible** tells if it's possible to further dissect a given flow. If the detected packet number is not enough and **ndpi_extra_dissection_possible** return true, nothing happen. Otherwise, the **detection_completed** will be set, and further packets belongs to this flow will not go through detection. Furthermore, **ndpi_detection_giveup** is invoke to finish the detection process, which determine the final protocol and is introduced in section 5.2.8.

ndpi_extra_dissection_possible is as below:

```
u_int8_t  ndpi_extra_dissection_possible () {
    u_int16_t  proto = flow->detected_protocol_stack[1] ? flow->detected_protocol_stack[1] : flow->
        detected_protocol_stack[0];

    switch(proto) {
    case NDPI_PROTOCOL_HTTP:
        if ((flow->host_server_name[0] == '\0') || (flow->http.response_status_code == 0))
            return(1);
        break;
    ...
    }
    return(0);
}
```

For most protocols, **extra_dissection_possible** directly return 0, means it is not necessary to detect extra packets. For other protocols, some condition must be satisfied before this function return 1. Take **HTTP** as an example, base on the previous discussion, even if the protocol is detected and determined, but still can not find the host name or the response code is still 0, nDPI will continue analyze later packets belong to this flow, until **enough_packets** is reached or this function return 0.

Back to the **processing_thread** function, the procedure of this function is summarized in Fig.2.

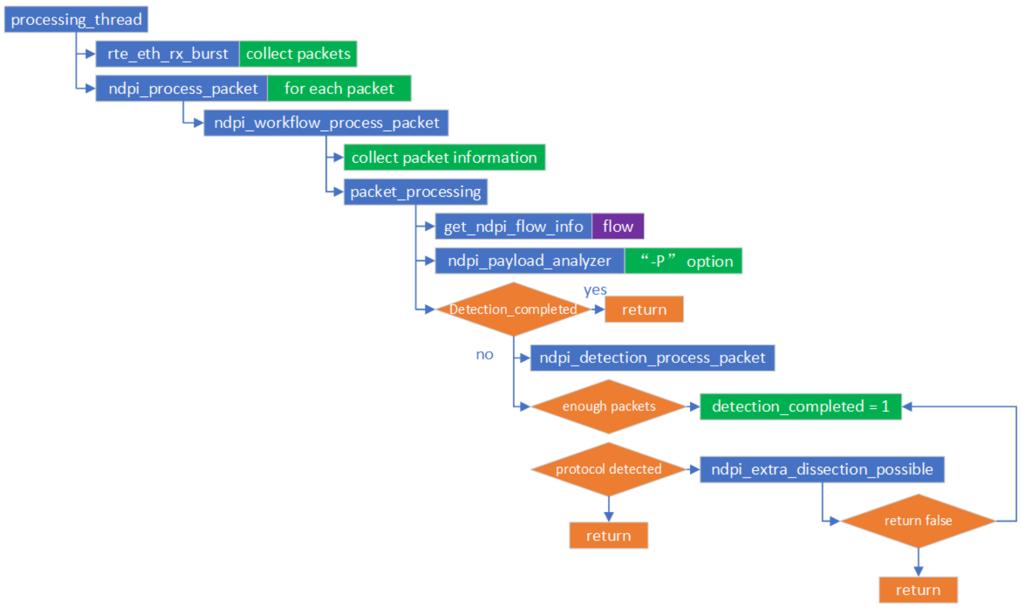


Figure 2: The procedure **processing_thread** function. As will depicted in other procedure diagrams, the blue box represents the function name; the green box is the interpretation; the purple box denotes the key structure; the orange diamond means the condition statement, furthermore, the right arrow of this diamond means the true result and the down arrow is the false result.

2.3.3 Print Result

As discussed in **test_lib** function, after initialization and processing all the packets, all the threads will exit, and statistic results will be print through **printResults** function, as below:

Listing 12: 1st Part of **printResults**

```
static void printResults(u_int64_t processing_time_usec, u_int64_t setup_time_usec) {

    memset(&cumulative_stats, 0, sizeof(cumulative_stats));

    for(thread_id = 0; thread_id < num_threads; thread_id++) {
        if((ndpi_thread_info[thread_id].workflow->stats.total_wire_bytes == 0)
                && (ndpi_thread_info[thread_id].workflow->stats.raw_packet_count == 0))
            continue;

        for(i=0; i<NUM_ROOTS; i++) {
            ndpi_twalk(ndpi_thread_info[thread_id].workflow->ndpi_flows_root[i],
                node_proto_guess_walker, &thread_id);
        }
```

cumulative_stats aggregates statistic of each thread. For each thread, check if **total_wire_bytes** and **raw_packet_count** are both 0, which means this thread didn't analyze any packet, just continue to the next thread. Otherwise, through the **ndpi_twalk** function, traversal each flow under the root node **ndpi_flows_root** to gather statistic, this function is introduced in section 3.1.3.

Note that the **total_wire_bytes** is calculated in **packet_processing** function, and **raw_packet_count** is increased in **ndpi_workflow_process_packet** function when receive a new packet.

Listing 13: 1st Part of **node_proto_guess_walker**

```
static void node_proto_guess_walker(const void *node, ndpi_VISIT which, int depth, void *user_data) {

    if((which == ndpi_preorder) || (which == ndpi_leaf)) {
        if((!flow->detection_completed) && flow->ndpi_flow) {
            u_int8_t proto_guessed;

            flow->detected_protocol = ndpi_detection_giveup(ndpi_thread_info[0].workflow->ndpi_struct
                , flow->ndpi_flow, enable_protocol_guess, &proto_guessed);
        }
```

As described in section 3.1.3. **ndpi_preorder** and **ndpi_leaf** specify the way of traversal. For each node(flow) on the **ndpi_flows_root** tree, first check if the detection is completed, if not, **ndpi_detection_giveup** should be used to determine the **detected_protocol** base on already guessed protocol ID.

Listing 14: 2nd Part of **node_proto_guess_walker**

```
        process_ndpi_collected_info(ndpi_thread_info[thread_id].workflow, flow, csv_fp);

        proto = flow->detected_protocol.app_protocol ? flow->detected_protocol.app_protocol : flow->
            detected_protocol.master_protocol;

        ndpi_thread_info[thread_id].workflow->stats.protocol_counter[proto]       += flow->
            src2dst_packets + flow->dst2src_packets;
        ndpi_thread_info[thread_id].workflow->stats.protocol_counter_bytes[proto] += flow->
            src2dst_bytes + flow->dst2src_bytes;
        ndpi_thread_info[thread_id].workflow->stats.protocol_flows[proto]++;
    }
}
```

Then **process_ndpi_collected_info** is invoked to collect some information and save to the **flow**. Finally, 3 counters is calculated. **protocol_counter** stands for total number of packets received of a specified protocol, **protocol_counter_bytes** is the total bytes and **protocol_flows** is the amount of flow.

Then, take a look at the **process_ndpi_collected_info** function:

Listing 15: 1st Part of **process_ndpi_collected_info**

```
void process_ndpi_collected_info(struct ndpi_workflow * workflow, struct ndpi_flow_info *flow, FILE *
    csv_fp) {
    ...
    else if((flow->detected_protocol.master_protocol == NDPI_PROTOCOL_HTTP)
            || is_ndpi_proto(flow, NDPI_PROTOCOL_HTTP)) {
        if(flow->ndpi_flow->http.url != NULL) {
            snprintf(flow->http.url, sizeof(flow->http.url), "%s", flow->ndpi_flow->http.url);
            flow->http.response_status_code = flow->ndpi_flow->http.response_status_code;
            snprintf(flow->http.content_type, sizeof(flow->http.content_type),
                    "%s", flow->ndpi_flow->http.content_type ? flow->ndpi_flow->http.content_type : "
                        ");
            snprintf(flow->http.user_agent, sizeof(flow->http.user_agent), "%s", flow->ndpi_flow->
                http.user_agent ? flow->ndpi_flow->http.user_agent : "");
        }
    }
    ...
}
```

This function collect the information relates to some protocols before cancel the detection, take HTTP as an example, before give up the detection, the URL, response code, content type, and agent will be determined.

Listing 16: 2nd Part of **printResults**

```
        cumulative_stats.raw_packet_count += ndpi_thread_info[thread_id].workflow->stats.
            raw_packet_count;
        cumulative_stats.ip_packet_count += ndpi_thread_info[thread_id].workflow->stats.
            ip_packet_count;
        cumulative_stats.total_wire_bytes += ndpi_thread_info[thread_id].workflow->stats.
            total_wire_bytes;
        cumulative_stats.total_ip_bytes += ndpi_thread_info[thread_id].workflow->stats.total_ip_bytes;
        cumulative_stats.total_discarded_bytes += ndpi_thread_info[thread_id].workflow->stats.
            total_discarded_bytes;

        for(i = 0; i < ndpi_get_num_supported_protocols(ndpi_thread_info[0].workflow->ndpi_struct); i
            ++) {
            cumulative_stats.protocol_counter[i] += ndpi_thread_info[thread_id].workflow->stats.
                protocol_counter[i];
            cumulative_stats.protocol_counter_bytes[i] += ndpi_thread_info[thread_id].workflow->stats
                .protocol_counter_bytes[i];
            cumulative_stats.protocol_flows[i] += ndpi_thread_info[thread_id].workflow->stats.
                protocol_flows[i];
        }
```

The second part of **printResults** function starts filling the **cumulative_stats** structure. Each item can be explained as below:

- **raw_packet_count**: The total packets received.
- **ip_packet_count**: The total IP packets received.
- **total_wire_bytes**: The total bytes received, includes the CRC(24bytes).
- **total_ip_bytes**: The total bytes received, excludes the CRC.
- **total_discarded_bytes**: The total bytes discarded, a packet can be discarded for many reason, such as bad CRC, bad FCS, or failed create a flow in **get_ndpi_flow_info** function, etc.

ndpi_get_num_supported_protocols return the number of supported protocols. The remaining **protocol_counter**, **protocol_counter_bytes** and **protocol_flows** have been discussed in the 2nd Part of **node_proto_guess_walker** function.

Listing 17: 3rd Part of **printResults**

```
            cumulative_stats.ndpi_flow_count += ndpi_thread_info[thread_id].workflow->stats.
                ndpi_flow_count;
            cumulative_stats.tcp_count    += ndpi_thread_info[thread_id].workflow->stats.tcp_count;
            cumulative_stats.udp_count    += ndpi_thread_info[thread_id].workflow->stats.udp_count;
            ...
            for(i = 0; i < sizeof(cumulative_stats.packet_len)/sizeof(cumulative_stats.packet_len[0]); i
                ++)
                cumulative_stats.packet_len[i] += ndpi_thread_info[thread_id].workflow->stats.packet_len[i
                    ];
            cumulative_stats.max_packet_len += ndpi_thread_info[thread_id].workflow->stats.
                max_packet_len;
        }

        ...
```

The third part still collects statistic base on some protocol:

- **ndpi_flow_count**: The total flow count, increase each time when successfully inserting an flow by **get_ndpi_flow_info** function.
- **tcp_count**: The received TCP packet amount, which is also set by **get_ndpi_flow_info** function.
- **udp_count**: The same as **tcp_count**, represents the total received UDP packet count.

After that, the packet length is calculated in **packet_len** array, the index in this array is determined by times between current packet length and the length of first **packet_len**, in most of time, it is 64.

Listing 18: 4th Part of **printResults**

```
        ...
        printf (...( unsigned long)cumulative_stats.tcp_count);
        ...
        if(processing_time_usec > 0) {
            ...
        }
```

The 4th part first print many statistics calculated in previous part, then, some throughput information is calculated and printed base on **processing_time_usec** value, which is the time interval begin when detection threads created and end when these threads exit.

Listing 19: 5th Part of **printResults**

```
        for(i = 0; i <= ndpi_get_num_supported_protocols(ndpi_thread_info[0].workflow->ndpi_struct); i
            ++) {
            if(cumulative_stats.protocol_counter[i] > 0) {
                printf (...,  ndpi_get_proto_name(ndpi_thread_info[0].workflow->ndpi_struct, i),
                    (long long unsigned int)cumulative_stats.protocol_counter[i],
                    (long long unsigned int)cumulative_stats.protocol_counter_bytes[i],
                    cumulative_stats.protocol_flows[i]);

                total_flow_bytes += cumulative_stats.protocol_counter_bytes[i];
            }
        }
```

For each supported protocol(includes the customer protocol by **"-p"**), if some packets is detected, and thus the corresponding item in **protocol_counter** array is non-zero, then, some information relates to this protocol is printed, includes protocol name, amount of packet, bytes and flow received.

Listing 20: 6th Part of **printResults**

```
        printFlowsStats();
         ...
    }
```

Finally, some flow statistic will be print in **printFlowsStats** function, such as port topology, ip distribution, payload stats, etc.

3 nDPI Algorithm

This section will introduce three data structure for searching the protocol:

1) **Binary Sort Tree:** used to index the port number to the protocol ID.

2) **Patricia Tree:** used to index the IP address to the protocol ID.

3) **AC Trie:** used to index the URL to the protocol ID.

Note that, as will introduce in section 4.1.3, the category will also use there structures, and save the category index instead of protocol ID.

3.1 Binary Sort Tree

A binary sort tree, which is much simple than the other two trees, has the property that for each 2 child node, the value in left child node is smaller than the right child node. nDPI use this tree to search the protocol ID base on the port value.

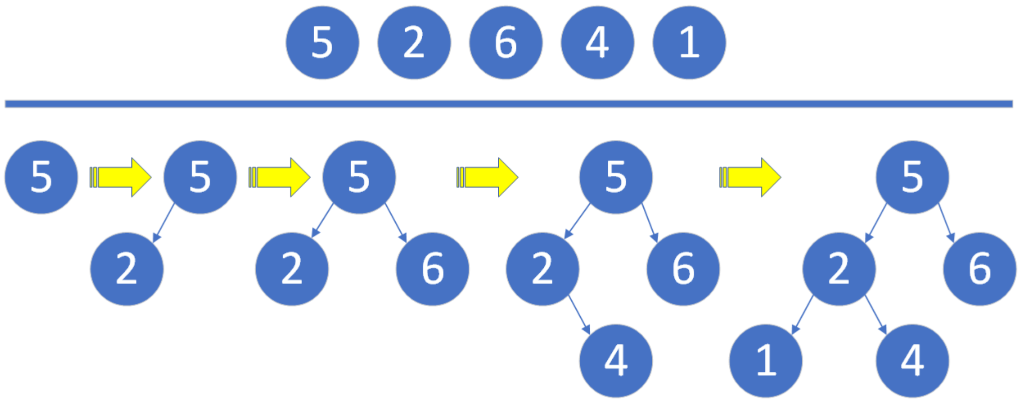


Figure 3: An Example of Binary Sort Tree. Construct by the nodes with values 5,2,6,4,1 in order. Below describes the procedure of generating the tree, base on the left child node's value is smaller than the right child node's value.

Fig.3 shows a binary sort tree, which has the property that for each 2 child node, the value in left child node is smaller than the right child node.

3.1.1 Source Code Analysis: Insert

The main insert function of binary sort tree is **ndpi_tsearch**, as below:

```
void * ndpi_tsearch(const void *vkey, void **vrootp, int (*compar)(const void *, const void *)) {
    ...

  while (*rootp != (ndpi_node *)0) {
      int r;

      if ((r = (*compar)(key, (*rootp)->key)) == 0)
          return ((*rootp)->key);
      rootp = (r < 0) ? &(*rootp)->left : &(*rootp)->right;
  }
  q = (ndpi_node *) ndpi_malloc(sizeof(ndpi_node));
  if (q != (ndpi_node *)0) {
      *rootp = q;
      q->key = key;
```

```
            q->left = q->right = (ndpi_node *)0;
        }
        return ((void *)q->key);
    }
```

This function can be concluded with following steps:

1) Search from the root node which pass from **ndpi_set_proto_defaults** function. It is either **tcpRoot** or **udpRoot**.

2) For each node, using the compare function to determined if match, if yes, return this node, otherwise, repeat step 2) on the child node, which is selected base on the return value of compare function.

3) If finally get to the leave node, and still mismatch, **ndpi_malloc** an new node and initialize the value, set the leave node as its parent node, insert into the tree.

3.1.2 Source Code Analysis: Search

The main search function of nDPI binary sort tree is **ndpi_tfind**, as below:

```
    void * ndpi_tfind(const void *vkey, void *vrootp, int (*compar)(const void *, const void *)) {
        char *key = (char *)vkey;
        ndpi_node **rootp = (ndpi_node **)vrootp;

        if(rootp == (ndpi_node **)0)
            return ((ndpi_node *)0);
        while (*rootp != (ndpi_node *)0) {
            int r;
            if((r = (*compar)(key, (*rootp)->key)) == 0)
                return (*rootp);
            rootp = (r < 0) ? &(*rootp)->left : &(*rootp)->right;
        }
        return (ndpi_node *)0;
    }
```

Similar as **ndpi_tsearch** function, from the root port, **ndpi_tfind** walk through the nodes to find a match node. The match function is the **compar**, which must be the same as the **compar** in the **ndpi_tsearch**.

3.1.3 Source Code Analysis: Walk

In some situation, there is a need to walk all the node in the binary sort tree, and handle each node with a function. This can be achieved by using **ndpi_twalk** function:

```
    void ndpi_twalk(const void *vroot, void (*action)(const void *, ndpi_VISIT, int, void *), void *
        user_data) {
        ndpi_node *root = (ndpi_node *)vroot;

        if(root != (ndpi_node *)0 && action != (void (*)(const void *, ndpi_VISIT, int, void*))0)
            ndpi_trecurse(root, action, 0, user_data);
}
```

The parameter **vroot** represent the root node of a binary sort tree, but it can also represents any node in the binary sort tree. For example, it called this function from node A, then all the child node of A will be traversal.

The second parameter **action** and the third parameter **user_data** is a callback function and corresponding paramter.

This function simply call the **ndpi_trecurse** as below:

```
static void ndpi_trecurse(ndpi_node *root, void (*action)(const void *, ndpi_VISIT, int, void*), int
    level, void *user_data) {
    if(root->left == (ndpi_node *)0 && root->right == (ndpi_node *)0)
        (*action)(root, ndpi_leaf, level, user_data);
    else {
        (*action)(root, ndpi_preorder, level, user_data);
        if(root->left != (ndpi_node *)0)
            ndpi_trecurse(root->left, action, level + 1, user_data);
        (*action)(root, ndpi_postorder, level, user_data);
        if(root->right != (ndpi_node *)0)
            ndpi_trecurse(root->right, action, level + 1, user_data);
        (*action)(root, ndpi_endorder, level, user_data);
    }
}
```

The steps can be summarized as below:

1) If current node **root** is the leave node, directly invoke the callback function and return.

2) Recurse the left child node then the right child node. The callback function **action** is called in 3 different position: tt the beginning, after recursing the left child node, or after recursing the right child node.

Base on the traversal method, to avoid walking the same node multiple times. These 3 functions can be selected by **ndpi_leaf**,**ndpi_preorder**,**ndpi_postorder** and **ndpi_endorder**.

3.2 Patricia Tree

A patricia tree is a special type of trie(prefix tree). A trie is something like a dictionary, each node contains a character, and a flag within current node to indicate if all the node from root to current node can generate an word, as show in below Fig.4.

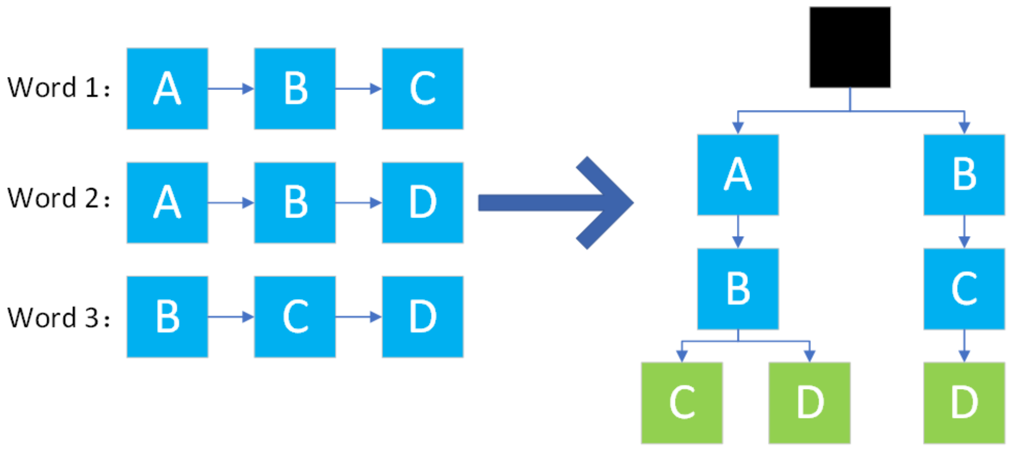


Figure 4: Three word generate an trie, black box represents the root of tree, and the final flag is true in the green box.

Notice that the black box in Fig.4 represents the root of tree, and in most of time, it is just an empty node.
On the other way, patricia tree is a **compact** trie, where two or more successive alone node are merged to one node. As in the Fig.5.

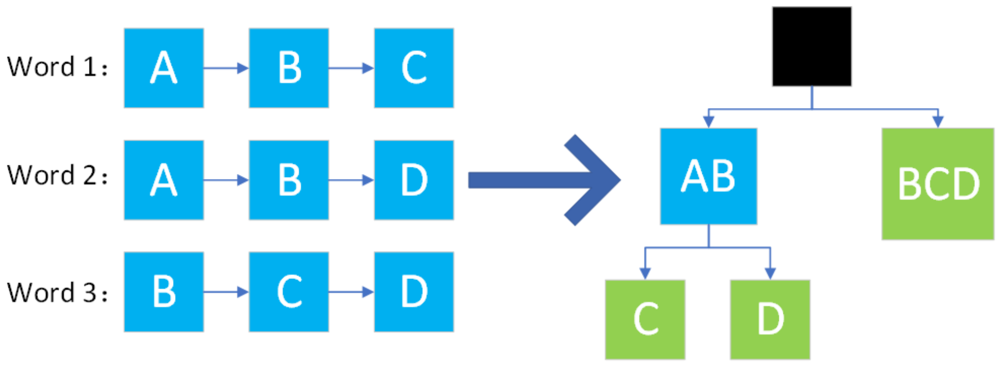


Figure 5: Three word generate an patricia tree, where some nodes are merged.

Comparing Fig.4 and Fig.5, in the left, **A**,**B** are merged as a single node **AB**; similarly, in the right, **B**,**C**,**D** are merged as a single node **BCD**.

In my option, the searching speed of patricia tree is faster than trie, but trie has low modification latency.

3.2.1 nDPI Patricia Tree

This section will transfer the standard patricia tree to nDPI patricia tree for storing IP address.

There are some properties in IP address, one of which, is the mask, which specify an IP range. For example, **1.1.1.1/32** can be matched in the patricia tree under a node with prefix and mask **1.1.1.0/24**.

After adding the mask, the new patricia tree becomes in Fig.6.

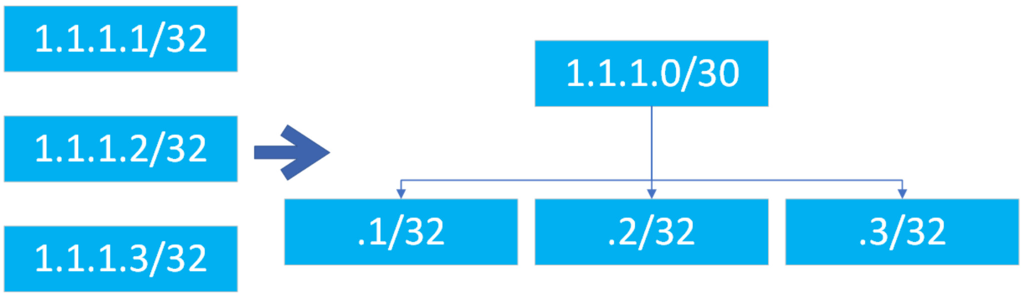


Figure 6: An example of patricia tree for IP address, adding the mask to each node.

After that, nDPI treat the whole IP prefix as binary, each node can only have at most 2 child nodes, base on the mask value and the prefix value on mask+1 position is 0 or 1. For example, **1.1.1.0/30** can have 2 child node, since the **30+1** bit is 0 and 1, respectively, one child node belongs to **1.1.1.0/31** range and the other belongs to **1.1.1.2/31** range. As in Fig 7.

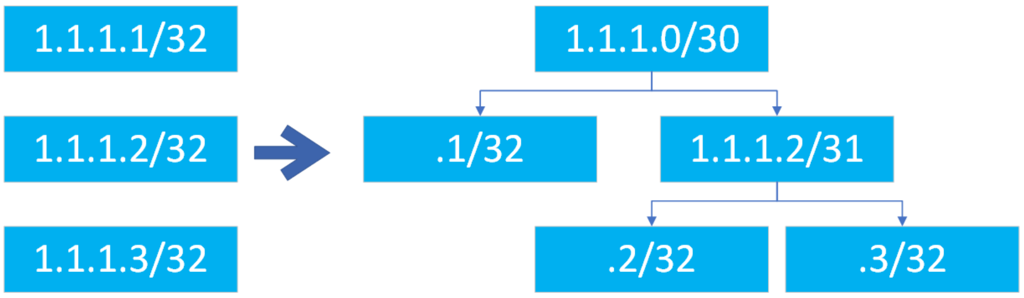


Figure 7: Limit the tree in Fig.6 to have only 2 child nodes, base on the prefix value on **mask+1**.

Then, nDPI optimize the structure in Fig.7 by migrate the whole IP prefix on the leave node. As in Fig.8.

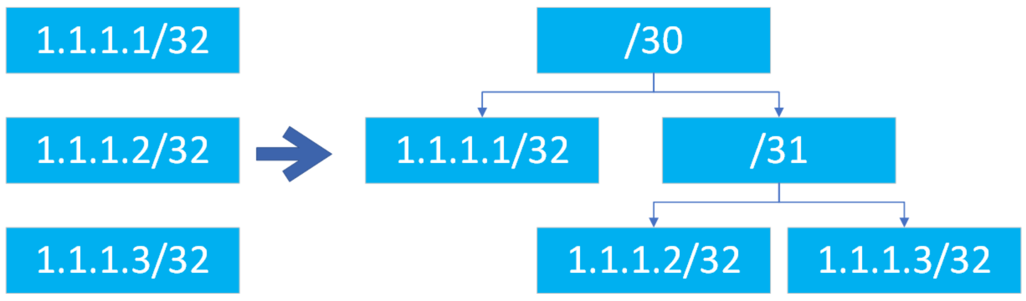


Figure 8: Migrate the whole IP prefix to the leave node.

The optimized patricia tree in above figure can reduce the comparing time when searching the tree. For example, if we want to lookup **1.1.1.3/32**, there are 3 match operation need to do, from **1.1.1.0/30** to **1.1.1.2/31** to **1.1.1.3/32**, with 3 different memory address. While in Fig.8, we just need to compare **1.1.1.3/32**. More detail will described in Section 3.2.3.

3.2.2 Example

Before deep into the source code, first take look as four example of patricia tree in nDPI. We take IP address as an example.

The first example is a patricia tree contains only one node, show in Fig.9.

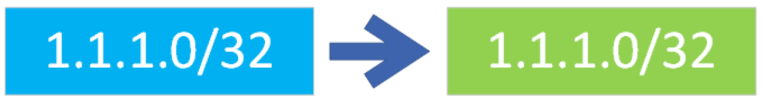


Figure 9: nDPI patricia tree with only one node.

The left part of slash is the IP prefix, the right part of the slash is the IP mask. After adding **"1.1.1.1/32"**to the tree. it becomes Fig.10.

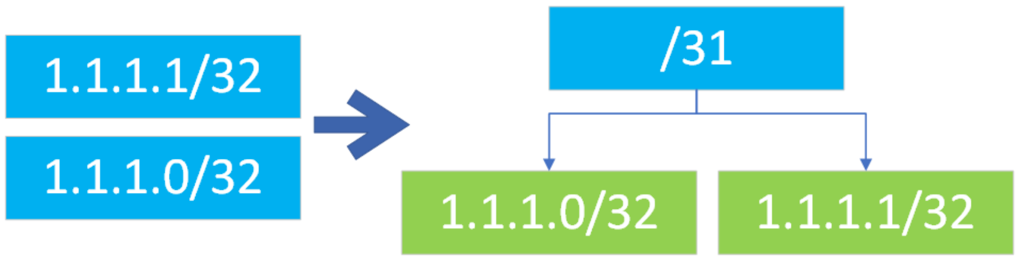


Figure 10: nDPI patricia tree with two node.

After adding the new node, the common components, which is **"1.1.1.0/31"** is extract as the new root node. And the original root node **"1.1.1.0/32"** now becomes the child node. It can be seen the new root node is an empty node, which only has a mask value. At this points, if adding another IP **"1.1.0.0/24"**, the patricia tree change to Fig.11.

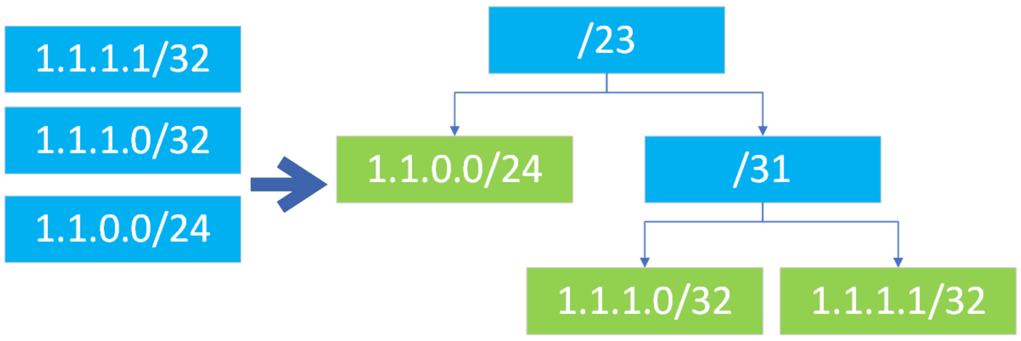


Figure 11: nDPI patricia tree with three node.

Now, a new root is generated with mask **23**, includes the left child node **"1.1.0.0/24"** and right child node **/31**. And the child node under node **/31** remains unchange.

After adding the last IP **"1.1.0.1/32"** to the tree, it finally turns into Fig.12.

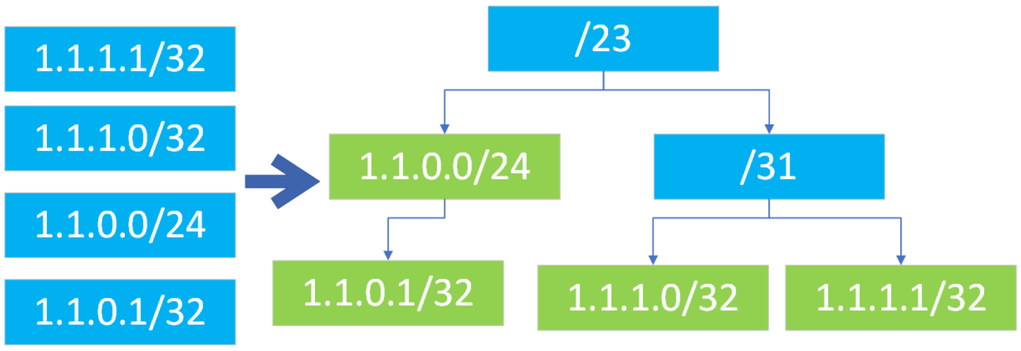


Figure 12: nDPI patricia tree with four node.

The new IP **"1.1.0.1/32"** is added as a child node of node **"1.1.0.0/24"**, and other part remains unchange.

3.2.3 Source Code Analysis: Create and Insert

An empty patricia tree can be allocated by using **ndpi_New_Patricia** function, shown as:

```
typedef struct _patricia_tree_t {
    patricia_node_t          *head;
    u_int16_t                maxbits;
    int num_active_node;
} patricia_tree_t ;

patricia_tree_t * ndpi_New_Patricia (u_int16_t maxbits) {
    patricia_tree_t *patricia = (patricia_tree_t *)ndpi_calloc(1, sizeof *patricia);

    patricia->maxbits = maxbits;
    patricia->head = NULL;
    patricia->num_active_node = 0;
    num_active_patricia++;
    return (patricia);
}
```

structure **patricia_tree_t** stores the global info of patricia tree:

- **head**: the head point of this patricia tree, one patricia tree can only have one head.

- **maxbits**: as described in section 3.2.1, nDPI treat the value on patricia three as binary, thus maxbits represent the maximum number of bits. For IPv4, it is fixed at 32.
- **num_active_node**: denotes how many nodes in the tree.

For and empty patricia tree, the **maxbits** is fixed, and the **head** is init to NULL, and **num_active_node** is 0.

Next part will introduce the procedure of adding an IP address into the patricia tree. The entry function is **add_to_ptree**, as below:

```
    static patricia_node_t *add_to_ptree( patricia_tree_t *tree, int family, void *addr, int bits) {
      prefix_t prefix;
      patricia_node_t *node;

      fill_prefix_v4 (&prefix, (struct in_addr *) addr, bits, tree->maxbits);

      node = ndpi_patricia_lookup(tree, &prefix);
      if(node)
          memset(&node->value, 0, sizeof(node->value));

      return(node);
  }
```

1) **fill_prefix_v4** fill the IP address **addr** and mask value **bits** to the **prefix** structure.

2) **ndpi_patricia_lookup** lookup the corresponding node in the tree, if not found, create one. Return the found or created node point.

3) If success, clear the **value** in the node, return the node point.

In above function, structure **patricia_node_t** denotes each node in the tree, defined as:

```
  typedef struct _patricia_node_t {
      u_int16_t bit;
      prefix_t *prefix;
      struct _patricia_node_t *l, *r;
      struct _patricia_node_t *parent;
      void *data;
      union patricia_node_value_t value;
  } patricia_node_t;
```

All the variables can be explain as:

bit: the IP mask.

prefix: the IP prefix.

l,r: the left and right child node point.

parent: the parent node, if current node is root, it is **NULL**.

data,value: stores the data value, such as a protocol ID.

Below is the main search and insert function **ndpi_patricia_lookup**, which is divided into several parts.

Listing 21: 1st part of **ndpi_patricia_lookup** function.

```
  patricia_node_t * ndpi_patricia_lookup (patricia_tree_t *patricia, prefix_t *prefix) {
      ...

      if(patricia->head == NULL) {
            node = (patricia_node_t*)ndpi_calloc(1, sizeof *node);
            node->bit = prefix->bitlen;
            node->prefix = ndpi_Ref_Prefix (prefix);
            node->parent = NULL;
            node->l = node->r = NULL;
            node->data = NULL;
            patricia->head = node;
            patricia->num_active_node++;
            return (node);
        }
```

Since the head point is **NULL** after create an empty patricia tree, as described in **ndpi_New_Patricia**, the if statement can only enter once when the first IP address is to be insert into the empty patricia tree.

Listing 22: 2nd part of **ndpi_patricia_lookup** function.

```
        addr = prefix_touchar (prefix);
        bitlen  = prefix->bitlen;
        node = patricia->head;

        while (node->bit < bitlen || node->prefix == NULL) {
            if (node->bit < patricia->maxbits &&
                    BIT_TEST (addr[node->bit >> 3], 0x80 >> (node->bit & 0x07))) {
                if (node->r == NULL)
                    break;
                node = node->r;
            } else {
                if (node->l == NULL)
                    break;
                node = node->l;
            }
        }
```

Variables **addr** and **bitlen** represent the IP prefix and mask. Note that for later comparison, the **prefix_touchar** function will reverse the order of the IP address. For example, **1.2.3.4** will be store in **addr** array as **addr[0]=1**,**addr[1]=2**,**addr[2]=3** and **addr[3]=4**. This part of code can be summarized as:

1) Traversal from the **head** of patricia tree.

2) If the mask in this node is smaller than the **bitlen**, or it is a glue node without prefix, enter the while.

3) Match the IP address with the node on the **mask+1** bit position. If match, goto the right child node, otherwise goto the left child node.

4) If the child node is **NULL**, exist the while.

In short, this part traversal the patricia tree, find a node with the minimum mask value and also match the IP address.

Listing 23: 3rd part of **ndpi_patricia_lookup** function.

```
        test_addr = prefix_touchar (node->prefix);
        check_bit = (node->bit < bitlen)? node->bit: bitlen;
         differ_bit  = 0;
        for (i = 0; (u_int)i*8 < check_bit; i++) {
            int r;

            if((r = (addr[i] ^ test_addr[i])) == 0) {
                differ_bit  = (i + 1) * 8;
                continue;
            }
            for (j = 0; j < 8; j++) {
                if(BIT_TEST (r, (0x80 >> j)))
                    break;
            }
             differ_bit  = i * 8 + j;
            break;
        }

        if( differ_bit  > check_bit)
             differ_bit  = check_bit;
```

After finding the node, this part calculates the first differ bits with the node. The first differ bit splits the IP address to two parts, the first part is match with the patricia tree, while the second part will be used to create an new node.

1) Take **check_bit** as the minimum length of the mask of IP address and the node's bits.

2) Compare a char(8bit) each time, if the result of xor is 0, continue match the next char, otherwise it means there are some bits different among the 8bit, goto next step to find it.

3) Compare each bit in the char we find, break if find the first different bit.

4) Now the **i** represents which index have the different bit, **j** means the different bit position in the char. So the **differ_bit** can be calculated.

5) Sometimes the **differ_bit** is larger than **check_bit**, for example, when **check_bit** < 8, which means the IP address is match with the node. Thus directly set the **differ_bit** to **check_bit**.

Listing 24: 4th part of **ndpi_patricia_lookup** function.

```
        parent = node->parent;
        while (parent && parent->bit >= differ_bit) {
            node = parent;
            parent = node->parent;
        }

        if( differ_bit  == bitlen && node->bit == bitlen) {
            if(node->prefix) {
                return (node);
            }
            node->prefix = ndpi_Ref_Prefix (prefix);
            return (node);
        }

        new_node = (patricia_node_t*)ndpi_calloc(1, sizeof *new_node);
        if (!new_node) return NULL;
        new_node->bit = prefix->bitlen;
        new_node->prefix = ndpi_Ref_Prefix (prefix);
        new_node->parent = NULL;
        new_node->l = new_node->r = NULL;
        new_node->data = NULL;
        patricia->num_active_node++;
```

Since in previous part of **ndpi_patricia_lookup_function** function, the leave node has been found, now it will traversal back to the root node, to find the exact position to insert the new IP address.

If the **differ_bit** equals to both the mask value of IP address and the node's bit, it means the node match exactly match the IP address has been found, then

1) If it have prefix value, which means this is a normal node, just return this node.

2) If it doesn't have the prefix value, means this is a glue node, set the prefix value to change the glue node to a normal node.

If the above procedure doesn't find the node, there will be **three** cases need to be handle in later code. Take Fig.12 as an example to explain how nDPI patricia tree handles these cases.

1st case:

The first if statement, denotes as

Listing 25: 5th part of **ndpi_patricia_lookup** function.

```
        if (node->bit == differ_bit) {
            new_node->parent = node;
            if (node->bit < patricia->maxbits && BIT_TEST (addr[node->bit >> 3], 0x80 >> (node->bit
                & 0x07))) {
                node->r = new_node;
            } else {
                node->l = new_node;
            }
            return (new_node);
        }
```

The first case is the node's bit equals to the **differ_bit**, but both are smaller than **bitlen**. This means the node only have one child, and the new IP address can be insert to the other child nodes. As depicts in Fig.13.

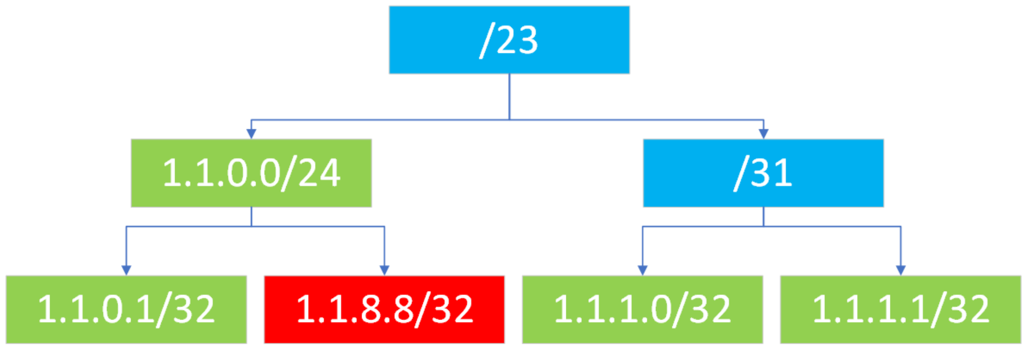


Figure 13: 1st base: add new node as child node.

2nd case:

The second if statement. Show up as:

Listing 26: 6th part of **ndpi_patricia_lookup** function.

```
        if ( bitlen  == differ_bit ) {
            if ( bitlen  < patricia->maxbits && BIT_TEST (test_addr[bitlen >> 3], 0x80 >> (bitlen & 0x07
               ))) {
                new_node->r = node;
            } else {
                new_node->l = node;
            }
            new_node->parent = node->parent;
            if (node->parent == NULL) {
                patricia->head = new_node;
            } else  if (node->parent->r == node) {
                node->parent->r = new_node;
            } else {
                node->parent->l = new_node;
            }
            node->parent = new_node;
        }
```

The second case is when the **bitlen** equals to the **differ_bit**, and both larger than the node's bit. It means the IP address is suitable to be the parent of the node. Then just add the new node as parent.

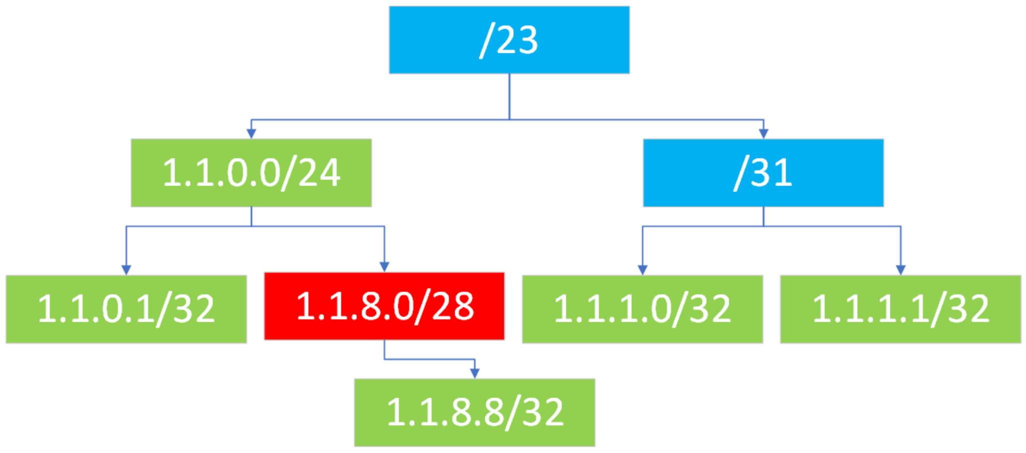


Figure 14: 2nd situation: add new node as parent node

3rd case:
the else statement, corresponding to the second part, with the source code as:

Listing 27: 7th part of **ndpi_patricia_lookup** function.

```
    else {
        glue = (patricia_node_t*)ndpi_calloc(1, sizeof *glue);
        glue->bit = differ_bit;
        glue->prefix = NULL;
        glue->parent = node->parent;
        glue->data = NULL;
        patricia->num_active_node++;
        if( differ_bit < patricia->maxbits && BIT_TEST (addr[differ_bit >> 3], 0x80 >> (differ_bit
            & 0x07))) {
            glue->r = new_node;
            glue->l = node;
        } else {
            glue->r = node;
            glue->l = new_node;
        }
        new_node->parent = glue;

        if(node->parent == NULL) {
            patricia->head = glue;
        } else if(node->parent->r == node) {
            node->parent->r = glue;
        } else {
            node->parent->l = glue;
        }
        node->parent = glue;
    }
    return (new_node);
}
```

In the third case, the **differ_bit** neither equals to the **bitlen** nor the node's bit. Then, a glue node without prefix will be created, which will be the parent of the new node and old child node. As depicts in Fig.15.

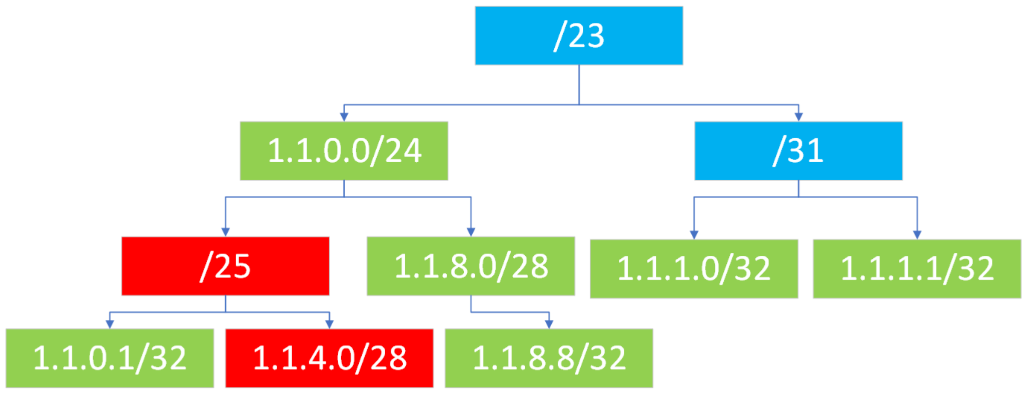


Figure 15: The 3rd case: add new node "1.1.4.0/28" under node "1.1.0.0/24", the old child node is "1.1.0.1/32", a glue node "/25" is created, and becomes a new parent of both new node and the old child node.

The step can be summarized as follow:

1) Create a glue node, whose parent is the old node's parent.

2) Add both new node and old node as the glue node's child.

3) If old node's parent is **NULL**, which means it is the head of patricia tree, then the new glue node becomes the head.

4) Add the glue node to it's parent.

3.2.4 Source Code Analysis: Search

nDPI only have two search functions for the patricia tree, one is **ndpi_patricia_search_best2**, the other is **ndpi_patricia_search_exact**.

While function **ndpi_patricia_search_exact** only search if an IP address is not in the patricia tree, it can't guarantee the IP address is in the tree.

This section only analyzes the first search function **ndpi_patricia_search_best2**, shown up as:

Listing 28: 1st part of **ndpi_patricia_search_best2**.

```
patricia_node_t * ndpi_patricia_search_best2 ( patricia_tree_t *patricia , prefix_t *prefix , int
    inclusive ) {
    ...

    node = patricia->head;
    addr = prefix_touchar (prefix);
    bitlen  = prefix->bitlen;

    while (node->bit < bitlen) {
        if (node->prefix) {
            stack[cnt++] = node;
        }

        if (BIT_TEST (addr[node->bit >> 3], 0x80 >> (node->bit & 0x07))) {
            node = node->r;
        } else {
            node = node->l;
        }

        if(node == NULL)
            break;
    }
```

This part traversal the patricia tree, the left child or the right child are also choose by the **mask+1** posetion in the **prefix**. Note that this function adds a stack to record all the nodes that have been pass through. The structure of this stack ensure later comparison will start from the largest mask value to the smallest mask value.

Listing 29: 2nd part of **ndpi_patricia_search_best2**.

```
    if( inclusive && node && node->prefix) {
        stack[cnt++] = node;
    }

    if(cnt <= 0)
        return (NULL);

    while (--cnt >= 0) {
        node = stack[cnt];
        if(ndpi_comp_with_mask (ndpi_prefix_tochar (node->prefix),
                             ndpi_prefix_tochar (prefix),
                             node->prefix->bitlen) && node->prefix->bitlen <= bitlen) {
            return (node);
        }
    }
    return (NULL);
}
```

Note that the **inclusive** flag indicates if we want to get the exact match value. For example, a patricia tree with **1.1.1.1/32** as one of its leave node, with parent node **1.1.1.0/24**. If the searching IP address is **1.1.1.1/32** and the **inclusive** flag is true, the leave node with IP address **1.1.1.1/32** will be returned. Otherwise, this search function will only return **1.1.1.0/24**.

The second part pop each node from the stack, using **ndpi_comp_with_mask** to do an exact match operation. The other condition is the node's bitlen must smaller than the bitlen of searching IP address, note this condition is right of the **ndpi_comp_with_mask** function, which means it will be check first.

3.3 Aho-Corasick algorithm

In computer science, the AhoCorasick algorithm is a string-searching algorithm invented by Alfred V. Aho and Margaret J. Corasick.[1]. It is a kind of dictionary-matching algorithm that locates elements of a finite set of strings (the "dictionary") within an input text. It matches all strings simultaneously. The complexity of the algorithm is linear in the length of the strings plus the length of the searched text plus the number of output matches. Note that because all matches are found, there can be a quadratic number of matches if every substring matches (e.g. dictionary = a, aa, aaa, aaaa and input string is aaaa).

Informally, the algorithm constructs a finite-state machine that resembles a trie with additional links between the various internal nodes. These extra internal links allow fast transitions between failed string matches (e.g. a search for cat in a trie that does not contain cat, but contains cart, and thus would fail at the node prefixed by ca), to other branches of the trie that share a common prefix (e.g., in the previous case, a branch for attribute might be the best lateral transition). This allows the automaton to transition between string matches without the need for backtracking.

When the string dictionary is known in advance (e.g. a computer virus database), the construction of the automaton can be performed once off-line and the compiled automaton stored for later use. In this case, its run time is linear in the length of the input plus the number of matched entries.

The AhoCorasick string-matching algorithm formed the basis of the original Unix command fgrep.

3.3.1 Example

An examples is depicts in Fig.16. A trie is generated by ac algorithm, base on four word **"he"**, **"hers"**, **"his"**, **"she"**.

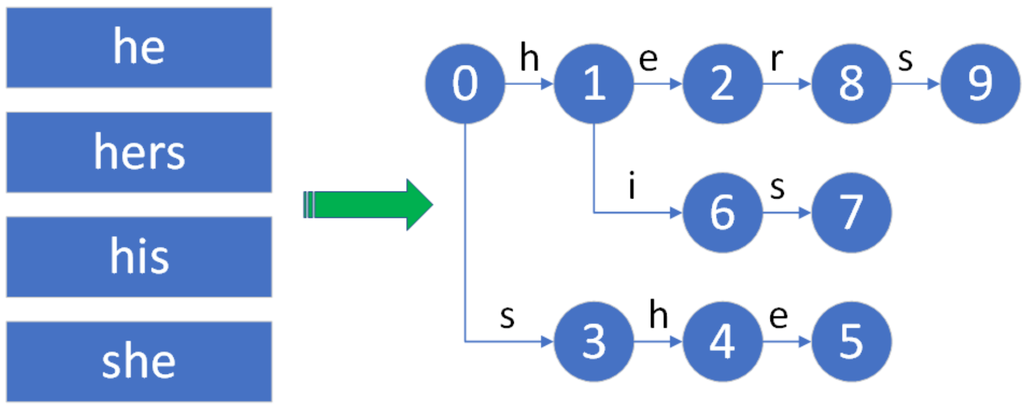


Figure 16: An Example of trie generated by AC algorithm.

As described in section 3.3, an ac trie is a special trie, which optimizes the normal trie in two ways:

1) Add extra internal links for failed string matches. As shown in the green dash line in Fig.17.

2) Add output for each match node. As shown in the red node in Fig.17.

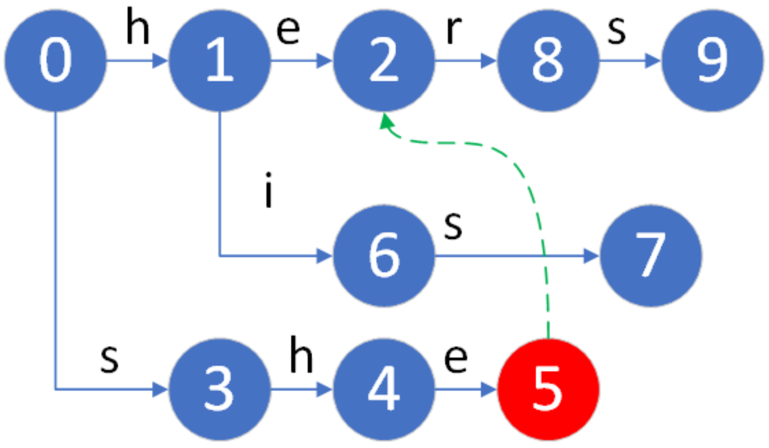


Figure 17: AC algorithm add internal links for failed string matches as green dash, also add output **"he"** on node 5.

The green dash and the red node also represents one of the fail function and output function in [1], respectively.

For example, if without the internal links, an string **"shers"** would go through node **0**, **3**, **4**, **5** to match the word **"she"**, meanwhile, go through node **0**, **1**, **2** to match the word **"he"**, and continue go through **8**, **9** to match word **"hers"**. And thus it need go through 9 node totally.

However, after ac trie adding the fail function and output function to the normal trie, it just needs to go through node **0**, **3**, **4**, **5**, **2**, **8**, **9** to match the word **"she"**, **"he"** and **"hers"**. Totally 7 nodes need to be visited, thus speedup the searching process.

3.3.2 Source Code Analysis: Structure

First take a look at some important structure relates to ac trie. **ac_node** denotes a node in the ac trie:

```c
typedef struct ac_node {
    short int final;
    struct ac_node * failure_node;
    unsigned short depth;

    AC_PATTERN_t * matched_patterns;
    unsigned short matched_patterns_num;
    unsigned short matched_patterns_max;

    struct edge * outgoing;
    unsigned short outgoing_degree;
    unsigned short outgoing_max;
} AC_NODE_t;
```

Some variables can be explained as:

- **final**: If this is a final node, means it contains the value, for nDPI, it is protocol ID.
- **failure_node**: The failure node of this node, if a mismatch happens on this node, it will jump to the failure node.
- **depth**: The distance between this node and the root.
- **outgoing**: Array of outgoing edges, each element in the array represents a character.
- **outgoing_degree**: Number of outgoing edges, i.e. the size of **outgoing** array.

Take Fig.17 as an example:

- For node 2, **final=0**, **failure_node=NULL**,**depth=2**, **outgoing="r"**, **outgoing_degree=1**.
- For node 5, **final=1**, **failure_node=2**, **depth=3**, **outgoing=NULL**,**outgoing_degree=0**

The other 3 variables **matched_patterns**, **matched_patterns_num** and **matched_patterns_max** is used mainly for debug purpose, and save the match results, or sometime act as an temp variable.
AC_PATTERN_t is defined as below:

```c
typedef struct {
    AC_ALPHABET_t * astring;
    unsigned int length;
    u_int8_t is_existing ;
    AC_REP_t rep;
} AC_PATTERN_t;
```

AC_ALPHABET_t is the same as **char**, some variables are explained below:

- **astring**: Represents the string, for example, "http://www.baidu.com".
- **length**: Length of the **astring**.
- **is_existing**: 1 if the node is already part of another **AC_PATTERN_t**, for example, suppose an **ac_node** contains 2 **matched_patterns**, one's **astring** is "aws.amazon.com", the other's **astring** is "ws.amazon.com", with **is_existing** set with 1.
- **rep**: Store the representative string, can be a protocol ID, and anything else.

AC_REP_t is define as:

```
typedef struct {
    u_int32_t  number;
    u_int16_t  category, breed;
} AC_REP_t;
```

In nDPI, **number** is used to store the protocol ID, **category** identifies the IP category, and breed relates to some security.

After all, all the information of a ac trie are represented using **AC_AUTOMATA_t** structure, defined as:

```
typedef struct {
    AC_NODE_t * root;
    AC_NODE_t ** all_nodes;

    unsigned int all_nodes_num;
    unsigned int all_nodes_max;

    AC_MATCH_t match;
    MATCH_CALLBACK_f match_callback;

    unsigned short automata_open;

    unsigned long total_patterns;

} AC_AUTOMATA_t;
```

Some variables are explained as:

- **root**: The root node of an trie, each search is start from the root node, thus each trie has only one root.
- **all_nodes**: Contains all the nodes in the trie.
- **match**: Act as an temp variables in searching process.
- **match_callback**: When successfully match an string, using this function to deal with the results, which stored in each node's **AC_REP_t** structure.
- **automata_open**: Identify if a trie is finalized by **ac_automata_finalize** or not, note that after finalizing, the ac trie is determined, no nodes can be added to this ac trie anymore.

3.3.3 Source Code Analysis: Init

An ac trie is built first through **ndpi_init_automa** function:

```
void *ndpi_init_automa(void) {
    return(ac_automata_init(ac_match_handler));
}
```

ac_match_handler is the callback function, when a node is match in searching process, this function will be call. It will be analyzed it in section 3.3.6.

```
AC_AUTOMATA_t * ac_automata_init (MATCH_CALLBACK_f mc) {
    AC_AUTOMATA_t * thiz = (AC_AUTOMATA_t *)ndpi_malloc(sizeof(AC_AUTOMATA_t));
    memset (thiz, 0, sizeof (AC_AUTOMATA_t));
    thiz->root = node_create ();
    thiz->all_nodes_max = REALLOC_CHUNK_ALLNODES;
    thiz->all_nodes = (AC_NODE_t **) ndpi_malloc (thiz->all_nodes_max*sizeof(AC_NODE_t *));
    thiz->match_callback = mc;
    ac_automata_register_nodeptr (thiz, thiz->root);
    thiz->total_patterns = 0;
```

```
        thiz->automata_open = 1;
        return thiz;
    }
```

First, a root node of the trie is created by **node_create**, the callback function is setup on **match_callback**. **ac_automata_register_nodeptr** is used to register the root node on the **all_nodes** variable as describe in **AC_AUTOMATA_t**. At last, **automata_open** is set to 1, means this ac trie can be added other nodes now.

3.3.4 Source Code Analysis: Insert

ndpi_string_to_automa is the entry of nDPI which adds a string to the ac trie, shown up as:

```
    static int ndpi_string_to_automa(struct ndpi_detection_module_struct *ndpi_str, ndpi_automa *automa,
        char *value,
                                  u_int16_t protocol_id, ndpi_protocol_category_t category,
                                      ndpi_protocol_breed_t breed,
                                  u_int8_t  free_str_on_duplicate ) {
        AC_PATTERN_t ac_pattern;
        ...

        ac_pattern.astring = value, ac_pattern.rep.number = protocol_id,
        ac_pattern.rep.category = (u_int16_t) category, ac_pattern.rep.breed = (u_int16_t) breed;
        ac_pattern.length = strlen(ac_pattern.astring);

        ac_automata_add(((AC_AUTOMATA_t *) automa->ac_automa), &ac_pattern);

        return(0);
    }
```

The **ac_pattern** acts as an temp variables, save 3 key variables: the **value** is the added string, i.e., an URL; the number records the guessed protocol ID relates to this URL; the length is the characters number of this URL.

After that, calling **ac_automata_add** to add **ac_pattern**, as below:

Listing 30: 1st Part of **ac_automata_add**

```
    AC_ERROR_t ac_automata_add (AC_AUTOMATA_t * thiz, AC_PATTERN_t * patt) {
        unsigned int i;
        AC_NODE_t * n = thiz->root;
        AC_NODE_t * next;
        AC_ALPHABET_t alpha;

        for (i=0; i<patt->length; i++) {
            alpha = patt->astring[i];
            if ((next = node_find_next(n, alpha))) {
                n = next;
                continue;
            } else {
                next = node_create_next(n, alpha);
                next->depth = n->depth + 1;
                n = next;
                ac_automata_register_nodeptr(thiz, n);
            }
        }
```

This function traversal each character in the **astring** from the root node, and do the following:

1) Using **node_find_next** to find the next node base on the current character. If found, repeat this step by using the next node.

2) If not find the node, create one, then repeat step 1) by using the created node.

ac_automata_register_nodeptr register the created node into the **all_nodes**.

```
AC_NODE_t * node_create_next (AC_NODE_t * thiz, AC_ALPHABET_t alpha) {
    AC_NODE_t * next;
    next = node_find_next (thiz, alpha);
    if (next)
        return NULL;
    next = node_create ();
    node_register_outgoing(thiz, next, alpha);

    return next;
}
```

node_create_next can be summarized as below:

1) confirm if the node is already under the parent node **thiz**, if true, return **NULL**.

2) If not find the node, create one, and register the edge information by using **node_register_outgoing** function, as below:

```
void node_register_outgoing(AC_NODE_t * thiz, AC_NODE_t * next, AC_ALPHABET_t alpha) {
    if(thiz->outgoing_degree >= thiz->outgoing_max) {
        thiz->outgoing = (struct edge *) ndpi_realloc(thiz->outgoing, thiz->outgoing_max*sizeof(
            struct edge),
            (REALLOC_CHUNK_OUTGOING+thiz->outgoing_max)*sizeof(struct edge));
        thiz->outgoing_max += REALLOC_CHUNK_OUTGOING;
    }

    thiz->outgoing[thiz->outgoing_degree].alpha = alpha;
    thiz->outgoing[thiz->outgoing_degree++].next = next;
}
```

The **outgoing** and related variables represent the edge information.Specifically, the **outgoing** array denotes the edge array of a node, and the **outgoing_degree** is the total edge number. If **outgoing_degree** is larger than **outgoing_max**, reallocate memory for store the new edge information. Then this function simply register the alpha and next node into the outgoing array.

Listing 31: 2nd Part of **ac_automata_add**

```
    if(n->final) {
        memcpy(&n->matched_patterns->rep, &patt->rep, sizeof(AC_REP_t));
        return ACERR_DUPLICATE_PATTERN;
    }

    n->final = 1;
    node_register_matchstr(n, patt, 0);
    thiz->total_patterns++;

    return ACERR_SUCCESS;
}
```

The second part of **ac_automata_add** first check if the final is true in the node, if it is, means the node has been allocated, thus just replace the representative **rep** and return; if not, set the **final** to 1, and register the **patt** in node, shown as:

```
void node_register_matchstr (AC_NODE_t * thiz, AC_PATTERN_t * str, u_int8_t is_existing) {
    ...
    if (thiz->matched_patterns_num >= thiz->matched_patterns_max) {
        ...
    }

    thiz->matched_patterns[thiz->matched_patterns_num].astring = str->astring;
    thiz->matched_patterns[thiz->matched_patterns_num].length = str->length;
    thiz->matched_patterns[thiz->matched_patterns_num].is_existing = is_existing;
    memcpy(&thiz->matched_patterns[thiz->matched_patterns_num].rep, &str->rep, sizeof(AC_REP_t))
        ;
    thiz->matched_patterns_num++;
}
```

Similar to the **node_register_outgoing** function, this function first checks if the memory is enough for storing the new pattern **str**, if not, allocate some memory for **matched_patterns**, which is just for debug purpose. The most important, is copy the representative information to the **rep**.

3.3.5 Source Code Analysis: Finalize

ac_automata_finalize is the entry point to finish the ac trie building process, as below:

```
void ac_automata_finalize (AC_AUTOMATA_t * thiz) {
    ...

    ac_automata_traverse_setfailure (thiz, thiz->root, alphas);

    for (i=0; i < thiz->all_nodes_num; i++) {
        node = thiz->all_nodes[i];
        ac_automata_union_matchstrs (node);
        node_sort_edges (node);
    }

    ...
}
```

There are two main task of this function:

One is **ac_automata_traverse_setfailure**, the other is **ac_automata_union_matchstrs**.

ac_automata_traverse_setfailure is the same as constructing the failure function in [1]. In nDPI, the failure function is corresponding to the **failure_node** which belongs to the structure **ac_node**.

ac_automata_union_matchstrs is the same as constructing the output function in [1]. In nDPI, the output function is corresponding to the **matched_patterns** array as described in the structure **ac_node**.

node_sort_edges sort the edge of each node base on the alpha value.

Construction of failure function:

The failure function is built by **ac_automata_traverse_setfailure**, as below

```
static void ac_automata_traverse_setfailure (AC_AUTOMATA_t * thiz, AC_NODE_t * node,
    AC_ALPHABET_t * alphas) {
    ...

    for (i=0; i < node->outgoing_degree; i++) {
        alphas[node->depth] = node->outgoing[i].alpha;
        next = node->outgoing[i].next;

        ac_automata_set_failure (thiz, next, alphas);

        ac_automata_traverse_setfailure (thiz, next, alphas);
        }
    }
```

This function using the depth-first-search(DFS) algorithm to walk through all node, the **alphas** array saves the character from root to the current node. **ac_automata_set_failure** set the failure node for current node(**next**), as below:

```
    static void ac_automata_set_failure (AC_AUTOMATA_t * thiz, AC_NODE_t * node,
        AC_ALPHABET_t * alphas) {
        ...

        for (i=1; i < node->depth; i++) {
            m = thiz->root;
            for (j=i; j < node->depth && m; j++)
                m = node_find_next (m, alphas[j]);
            if (m) {
                node->failure_node = m;
                break;
            }
        }

        if (!node->failure_node)
            node->failure_node = thiz->root;
    }
```

This function find the corresponding failure node for a node. Take Fig.17 as an example. Consider 4 cases:

Case 1: calculate failure_node on node 1 and node 3
Since the **depth**(explained in structure **struct_ac_node**) in both node 1 and node 3 is 1, the **failure_node** in this two nodes will be set to root node, means the state will be start from beginning.

Case 2: calculate failure_node on node 2
The **depth** in node 2 is 2, thus external **for** and internal **for** both will be execute only once. Consider the recursion properties of function **ac_automata_traverse_setfailure**, in this case alphas[0] would be **"s"** and alphas[1] would be **"r"**, but the root node does not have edge with alpha **"r"**, so the **failure_node** in this node still be set to root node.

Case 3: calculate failure_node on node 4
The **depth** in node 4 is also 2, thus external **for** and internal **for** both will be execute only once, in this case **alphas[0]** would be **"s"** and alphas[1] would be **"h"**, according to Fig.17, **node_find_next** would finally return node 1, and thus the **failure_node** in this node would be set to node 1.

Case 4: calculate failure_node on node 5
The **depth** in node 5 is 3, thus external **for** and internal **for** both will be executed up to twice(maybe only once). In this case **alphas[0]** would be **"s"**, **alphas[1]** would be **"h"** and **alphas[2]** would be **"e"**.

In the first external **for**, the internal **for** would be execute twice, according to the value in **alphas[1]** and **alphas[2]**, **m** will be node 2 in the end, and break the external **for**.
Repeat all this procedure, all the failure node is depicts in Fig.18.

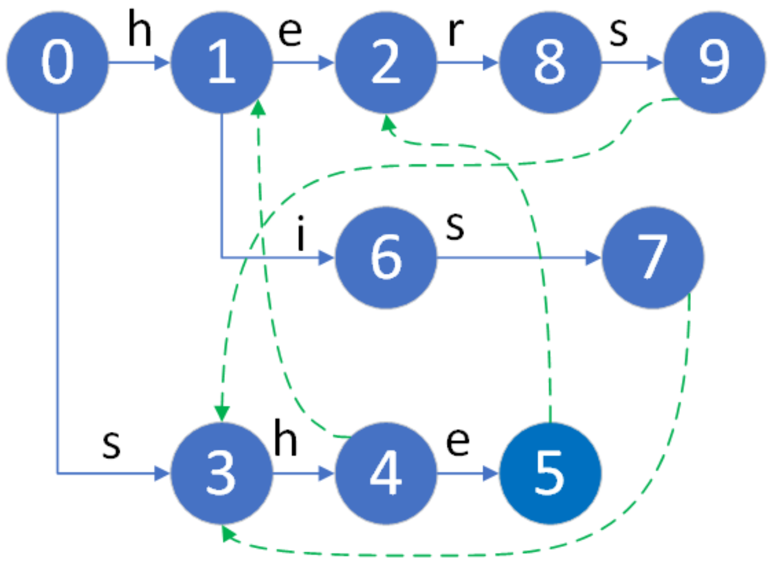


Figure 18: The result after executing **ac_automata_traverse_setfailure** function.

In short, we summarize the algorithm to set the failure node as:

If node B to node A with the edge array is equals to the root node to node C, than the node A's failure node will be set to node C.

Construction of output function:

The output function is built by **ac_automata_union_matchstrs** function, as below:

```
static void ac_automata_union_matchstrs (AC_NODE_t * node) {
    unsigned int i;
    AC_NODE_t * m = node;

    while ((m = m->failure_node)) {
        for (i=0; i < m->matched_patterns_num; i++)
            node_register_matchstr(node, &(m->matched_patterns[i]), 1);

        if (m->final)
            node->final = 1;
    }
}
```

Suppose the parameter is node A, the steps are:

1) Track all the failure node of node A, if the failure node doesn't exist, stop.

2) Traversal all the output function of each failure node.

3) Add all the output **matched_patterns** to node A.

To further explain this procedure, suppose constructing an ac trie base on 4 words, **"e"**(this is not a word, just take an example),**"he"**, **"hers"** and **"she"**, as depicts in Fig.19,

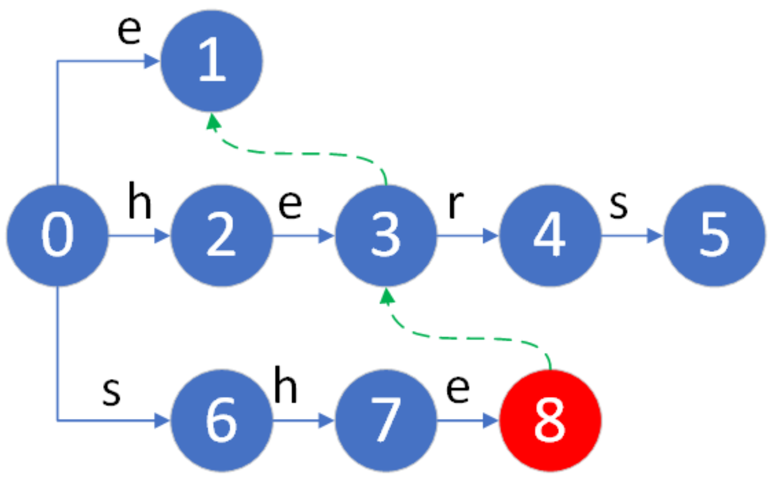


Figure 19: Add output in node **8**.

Node 8 has output **"she"** originally. After the node 8 enter **ac_automata_union_matchstrs** function, the first failure node 3 will add output **"he"** to node 8, the second failure node 1 will add output **"e"** to node 8.

Finally, all the edge of each node will be sort using **node_sort_edges** function:

```
void node_sort_edges (AC_NODE_t * thiz) {
    sort ((void *)thiz->outgoing, thiz->outgoing_degree, sizeof(struct edge), node_edge_compare,
        NULL);
}
```

node_edge_compare is the main compare function, and the affect of the sequence has be explained in Section.3.3.6.

3.3.6 Source Code Analysis: Search

ac_automata_search is the entry of searching a node in ac trie. As below:

```
int ac_automata_search (AC_AUTOMATA_t * thiz, AC_TEXT_t * txt, AC_REP_t * param) {
    ...

    s.current_node = thiz->root;
    position = 0;
    curr = s.current_node;

    while (position < txt->length) {
        if (!(next = node_findbs_next(curr, txt->astring[position]))) {
            if(curr->failure_node)
                curr = curr->failure_node;
            else
                position++;
        } else {
            curr = next;
            position++;
        }

        if(curr->final && next) {
            thiz->match.position = position;
            thiz->match.match_num = curr->matched_patterns_num;
```

```
                thiz->match.patterns = curr->matched_patterns;
                if (thiz->match_callback(&thiz->match, txt, param))
                    return 1;
            }
        }

        return 0;
    }
```

The steps of find the node in ac trie corresponding to the string **txt** is as below:

1) The **position** represents the index in the string **txt**, if it reaches the end of string, but yet find the node, return 0.

2) From the current node(first time it is the root node), for each character in the string **txt**, find the next node.

3) If failed to find the next node, check if current node has the failure node, which set in Fig.18 as green dash line. If current node has the failure node, replace the current node with the failure node, go back to step 2). Otherwise, go back to step 2) and using the next character in **txt** to find next node.

4) From step 2), if successfully find the next node, replace the current node with the next node.

5) If the **final** flag in current node is set and it is not a failure node, the **match_callback** would be called.

match_callback is set in **ac_automata_init** function, which is **ac_match_handler**.

Listing 32: 1st Part of **ac_match_handler**

```
static int ac_match_handler(AC_MATCH_t *m, AC_TEXT_t *txt, AC_REP_t *match) {
    int min_len = (txt->length < m->patterns->length) ? txt->length : m->patterns->length;
    char buf[64] = {'\0'}, *whatfound;
    int min_buf_len = (txt->length > 63) ? 63 : txt->length;

    strncpy(buf, txt->astring, min_buf_len);
    buf[min_buf_len] = '\0';

    whatfound = strstr(buf, m->patterns->astring);

    if(whatfound) {
        if((whatfound != buf) && (m->patterns->astring[0] != '.')
                && strchr(m->patterns->astring, '.')) {
            int len = strlen(m->patterns->astring);

            if((whatfound[-1] != '.') || ((m->patterns->astring[len - 1] != '.') && (whatfound[len]
                != '\0'))) {
                return(0);
            } else {
                memcpy(match, &m->patterns[0].rep, sizeof(AC_REP_t)); /* Partial match? */
                return(0);
            }
        }
    }
```

In the first part, some temp variables are initialized, and it also considers the case when the length of **txt** is larger than the size of **buf**, we don't consider this case for simply explaining this function.

We focus on the **whatfound** variable, which means if the prefix exist between the test string **txt**, and the node's string defined in **astring**, for example, if **txt** is "abc" and **astring** is "bc", then **whatfound** would be 1.

The steps when **whatfound** is nonzero can be summarized as below:

1) Check if **astring[0]** is “.”, if true, it's a match, jump to the second part of this function, otherwise, goto next step.

2) If **whatfound** do not start with “.”, denotes this is a mismatch. Otherwise, goto next step.

3) It is a match, then copy the representative **rep** and return 0.

Note that when the callback function **ac_match_handler** returns 1, it means there is no need to continue the searching process, so just finish the whole searching process. When return 0, it denotes that there is a possible match, but the searching process should be continue to find more precise node.

Listing 33: 2nd Part of **ac_match_handler**

```
        memcpy(match, &m->patterns[0].rep, sizeof(AC_REP_t));

        if (strncmp(buf, m->patterns->astring, min_len) == 0) {
            return(1);
        } else {
            return(0);
        }
    }
```

When entering this part of code, it means either the prefix of both **txt** and **astring** is the same, or the prefix are different but the first character of **astring** is “.”. In both cases, the representative **rep** will be copy, in the first case, this function returns 1; in the second case, it will return 0.

To clearly explain the searching process, consider 6 cases:

Case 1: txt is “www.baidu.com/sport”, trie contains a node with astring “www.baidu.com”.

In this case, after traversal the **txt** until the “m”, the function **ac_automata_search** will enter step 5. Then, since the prefix is the same, the second part of function **ac_match_handler** would be called, and finally return 1.

In short, in this case, it is a match, and the searching process stop.

Case 2: txt is “www.baidu.com”, trie contains a node with astring “www.baidu.com/sport”.

The same as case 1, since the second part of function **ac_match_handler** only compare the minimum length of **txt** and **astring**, in this case, it is a match, and the searching process stop.

Case 3: txt is “www.cccbaidu.com”, trie contains a node with astring “www.baidu.com”.

In this case, after traversal the **txt** until the “m”, the function **ac_automata_search** will enter step 5. The difference between this case and case 1 is that, before enter the callback function, step will repeat 3 times when go through the 3 “c” characters.

After that, the second part of function **ac_match_handler** would be called, and finally return 0.

In short, in this case, it is a match, and the searching process must continue.

Case 4: txt is “kkwww.baidu.com”, trie contains a node with astring “www.baidu.com”.

In this case, after traversal the **txt** until the “m”, the function **ac_automata_search** will enter step 5. The difference is also the same as case 3, which skip some comparison when go through the first “k” characters.

In **ac_match_handler** function, after calculating, since **buf** would be “kkwww.baidu.com”, and the **astring** is “www.baidu.com”, then **whatfound** would be 2, **whatfound[-1]** would be “k” and not equal to “.”, Thus, it will return 0.

In short, in this case, it is a mismatch, and the searching process must continue.

Case 5: txt is “kk.www.baidu.com”, trie contains a node with astring “www.baidu.com”.

The same as case 4, except that **whatfound[-1]** equals to “.”, and thus return 1.

In short, in this case, it is a match, and the searching process must continue.

Case 6: txt is “www.baidu.com/sport”, suppose trie contains two final nodes, one is node A with astring “www.ccc.baidu.com” and the other final node B contains “.baidu.com/sports”, from section 3.3.5, node B is the failure node of node A on “/” character.

In this case, node A will be first match since “www” is match first, since the “ccc” characters, according to case 3, it is a match, but the searching process continue. After that, it will jump to failure node B, base on the analysis of case 5, node B will replace node A and become the finally match node.

3.4 Summary

In this section, three data structure is introduced. One is binary sort tree, which uses port number as an index to search the protocol ID or category; the other is patricia tree, generated by IP address and mask; the last ac trie is used to store the URL information.

4 nDPI Initialization

Base on the introduction in section 2.3, in this section, some initialization function in nDPI libs will be introduced.

4.1 Workflow Initialization

As discussed in section 2.3.1, **ndpi_workflow_init** is the main function of nDPI initialization, shown as:

```
struct ndpi_workflow* ndpi_workflow_init(const struct ndpi_workflow_prefs * prefs,
                                         pcap_t * pcap_handle) {
    struct ndpi_detection_module_struct * module;
    struct ndpi_workflow * workflow;

    set_ndpi_malloc(ndpi_malloc_wrapper), set_ndpi_free(free_wrapper);
    set_ndpi_flow_malloc(NULL), set_ndpi_flow_free(NULL);

    module = ndpi_init_detection_module(ndpi_no_prefs);

    workflow = ndpi_calloc(1, sizeof(struct ndpi_workflow));
    workflow->pcap_handle = pcap_handle;
    workflow->prefs       = *prefs;
    workflow->ndpi_struct = module;

    workflow->ndpi_flows_root = ndpi_calloc(workflow->prefs.num_roots, sizeof(void *));

    return workflow;
}
```

First, function **set_ndpi_malloc** and **set_ndpi_free** is used to setup the memory allocate and free function, as below:

```
void set_ndpi_malloc(void *(*__ndpi_malloc)(size_t size)) {
    _ndpi_malloc = __ndpi_malloc;
}
```

When the **ndpi_malloc** is called, function **ndpi_malloc_wrapper** or the default **malloc** function in glibc will be called instead. Similar to **ndpi_free** function. They will be introduced in section 10.

Second, the same as **set_ndpi_malloc** function, **set_ndpi_flow_malloc** setup the flow allocate function, as below:

```
void set_ndpi_flow_malloc(void *(*__ndpi_flow_malloc)(size_t size)) {
    _ndpi_flow_malloc = __ndpi_flow_malloc;
}
```

If not specify this function, as in **ndpi_workflow_init** function which pass **NULL** to this function, the default allocate function would be **ndpi_malloc**, and the default free function would be **ndpi_free**.

Third, the main initialized function **ndpi_init_detection_module** would be called, which setup the **ndpi_detection_module_struct** structure mainly used for detection.

Finally, **ndpi_workflow** is initialized and return.

The main detection structure **ndpi_detection_module_struct** is initialized as:

Listing 34: 1st Part of **ndpi_init_detection_module**

```
{struct ndpi_detection_module_struct *ndpi_init_detection_module(ndpi_init_prefs prefs) {
    struct ndpi_detection_module_struct *ndpi_str = ndpi_malloc(sizeof(struct
        ndpi_detection_module_struct));
    int i;

```

```
        if ((ndpi_str->protocols_ptree = ndpi_New_Patricia(32)) != NULL)
            ndpi_init_ptree_ipv4 (ndpi_str, ndpi_str->protocols_ptree, host_protocol_list , prefs &
                ndpi_dont_load_tor_hosts);

        ...
```

ndpi_init_detection_module function first allocates a **ndpi_detection_module_struct** structure. Then, **ndpi_init_ptree_ipv4** function is called to construct the IP patricia tree for searching protocol ID base on IP address, below is the default IP to protocol mapping array:

```
    static ndpi_network host_protocol_list [] = {
        { 0x22FB2FEE /* 34.251.47.238 */, 32, NDPI_PROTOCOL_SOUNDCLOUD },
        { 0x23A06456 /* 35.160.100.86 */, 32, NDPI_PROTOCOL_SOUNDCLOUD },
        { 0x36C0CA58 /* 54.192.202.88 */, 32, NDPI_PROTOCOL_SOUNDCLOUD },

        ...
    }
```

Three items in each line is IP address, mask value and protocol, respectively.

ndpi_init_ptree_ipv4 is also called by ndpi_init_detection_module function. It will construct an patricia tree base on the IP address.

```
    {static void ndpi_init_ptree_ipv4 (struct ndpi_detection_module_struct *ndpi_str,
                                      void *ptree, ndpi_network host_list [], u_int8_t skip_tor_hosts) {
        for (i = 0; host_list [i].network != 0x0; i++) {
            struct in_addr pin;
            patricia_node_t *node;

            if (skip_tor_hosts && (host_list[i].value == NDPI_PROTOCOL_TOR))
                continue;

            pin.s_addr = htonl( host_list [i].network);
            if ((node = add_to_ptree(ptree, AF_INET, &pin, host_list[i].cidr)) != NULL) {
                node->value.uv.user_value = host_list[i].value, node->value.uv.additional_user_value = 0;
            }
        }
    }
```

This function traversal the **host_list**, add each IP address, mask value and protocol id to the patricia tree **ptree**, more details about **add_to_ptree** function have been introduced in section 3.2.3.

Listing 35: 2nd Part of **ndpi_init_detection_module**

```
        ndpi_str->host_automa.ac_automa = ac_automata_init(ac_match_handler);
        ndpi_str->content_automa.ac_automa = ac_automata_init(ac_match_handler);
        ...

        ndpi_init_protocol_defaults (ndpi_str);

        return(ndpi_str);
    }
```

The second part using **ac_automata_init** function to initialize 8 ac trie, this function has been analyzed in section 3.3.3. Then, **ndpi_init_protocol_defaults** will fill this ac trie with default information.

Some important variables are shown in picture below:

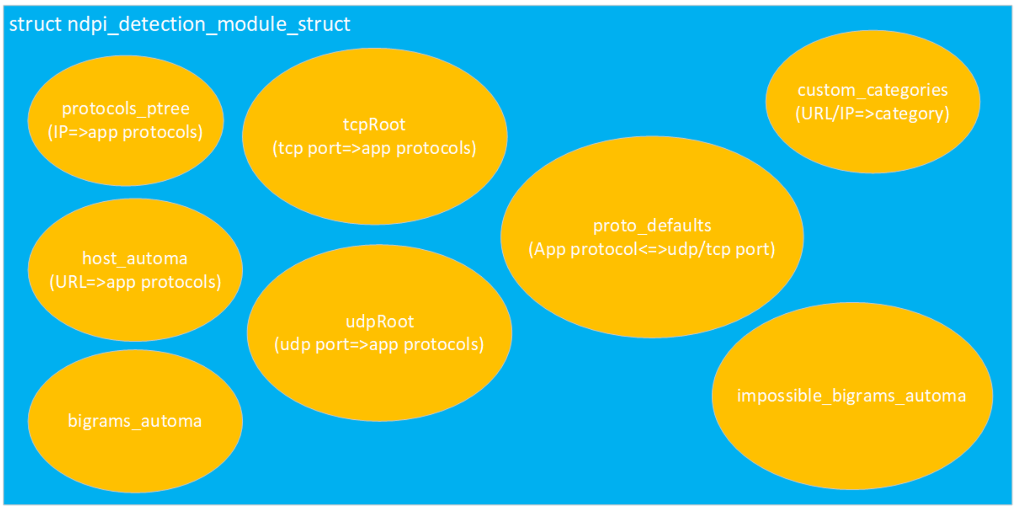

struct ndpi_detection_module_struct
protocols_ptree (IP=>app protocols)
tcpRoot (tcp port=>app protocols)
custom_categories (URL/IP=>category)
proto_defaults (App protocol<=>udp/tcp port)
host_automa (URL=>app protocols)
udpRoot (udp port=>app protocols)
impossible_bigrams_automa
bigrams_automa

Figure 20: The key entries of struct **ndpi_detection_module_struct(ndpi_str)**.

ndpi_workflow_init can be depict in Fig.21.

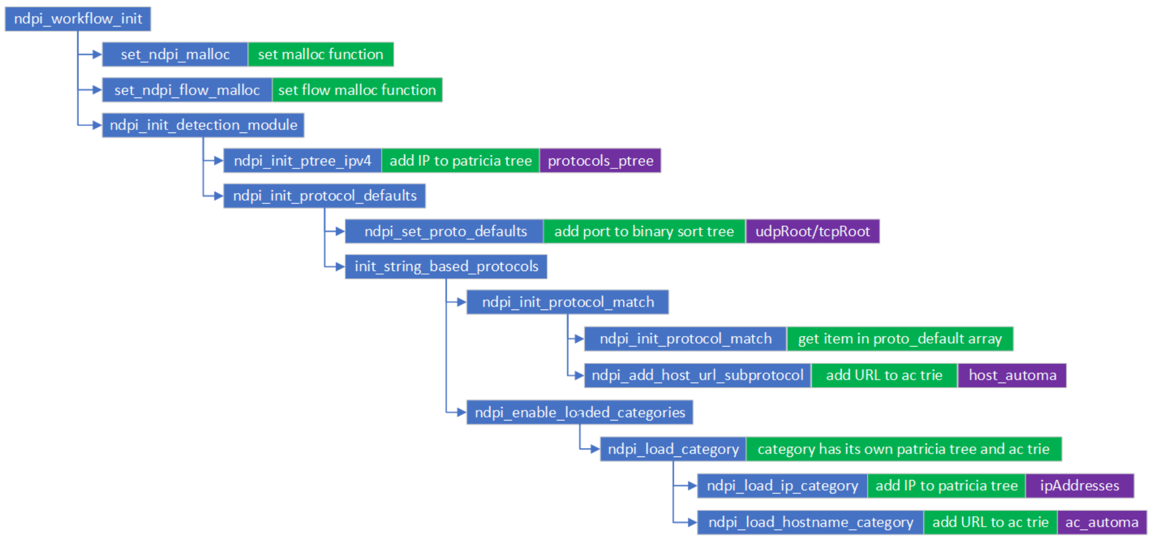


Figure 21: The procedure of **ndpi_workflow_init**. Note that, the green box is the explanation of the function, and purple box is the name of tree.

More details about the functions in Fig.21 will be analyzed in sub-sections.

4.1.1 Default Protocol Initilization

nDPI defines many default relationships between IP address, port number, URL and protocol ID. The main body of **ndpi_init_protocol_defaults** function defines some protocols with the relationship to port number, as below:

```
static void ndpi_init_protocol_defaults(struct ndpi_detection_module_struct *ndpi_str) {
    ndpi_port_range ports_a[MAX_DEFAULT_PORTS], ports_b[MAX_DEFAULT_PORTS];
    u_int16_t no_master[2] = {NDPI_PROTOCOL_NO_MASTER_PROTO,
        NDPI_PROTOCOL_NO_MASTER_PROTO}, custom_master[2];

    memset(ndpi_str->proto_defaults, 0, sizeof(ndpi_str->proto_defaults));

```

```
        ndpi_set_proto_defaults(ndpi_str, NDPI_PROTOCOL_UNRATED,
            NDPI_PROTOCOL_UNKNOWN, 0,
                                no_master, no_master, "Unknown",
                                    NDPI_PROTOCOL_CATEGORY_UNSPECIFIED,
                                ndpi_build_default_ports(ports_a, 0, 0, 0, 0, 0) /* TCP */,
                                ndpi_build_default_ports(ports_b, 0, 0, 0, 0, 0) /* UDP */);
        ndpi_set_proto_defaults(ndpi_str, NDPI_PROTOCOL_UNSAFE,
            NDPI_PROTOCOL_FTP_CONTROL, 0,
                                no_master, no_master, "FTP_CONTROL",
                                    NDPI_PROTOCOL_CATEGORY_DOWNLOAD_FT,
                                ndpi_build_default_ports(ports_a, 21, 0, 0, 0, 0) /* TCP */,
                                ndpi_build_default_ports(ports_b, 0, 0, 0, 0, 0) /* UDP */);

        ...

        init_string_based_protocols(ndpi_str);

         ndpi_validate_protocol_initialization(ndpi_str);
    }
```

ndpi_set_proto_defaults is adding the port and protocol ID relationship into the binary sort tree. It can be seen that there are at most five tcp and udp ports corresponding to a protocol, they can generate a range variable through **ndpi_build_default_ports** function, but only one information will be store in the **proto_defaults** for each protocol.

ndpi_set_proto_defaults is as below:

Listing 36: 1st part of **ndpi_set_proto_defaults**

```
    void ndpi_set_proto_defaults(struct ndpi_detection_module_struct *ndpi_str,
            ndpi_protocol_breed_t breed, u_int16_t protoId, u_int8_t can_have_a_subprotocol,
            ... char *protoName, ndpi_protocol_category_t protoCategory,
            ndpi_port_range *tcpDefPorts,ndpi_port_range *udpDefPorts) {

        ...

        ndpi_str->proto_defaults[protoId].protoName = name;
        ndpi_str->proto_defaults[protoId].protoCategory = protoCategory;
        ndpi_str->proto_defaults[protoId].protoId = protoId;
        ndpi_str->proto_defaults[protoId].protoBreed = breed;
        ndpi_str->proto_defaults[protoId].can_have_a_subprotocol = can_have_a_subprotocol;

        ...
```

This function set the pre-defined value in **ndpi_init_protocol_defaults** function. Some value is obvious, we focus on parts of this value.

- **can_have_a_subprotocol**: this value specify if a protocol can have a sub-protocol, for example, a HTTP protocol maybe have Skype sub-protocol.
- **protoBreed**: this value is something relates to the security in nDPI. It may be introduced in future.

Listing 37: 2nd part of **ndpi_set_proto_defaults**

```
        for (j = 0; j < MAX_DEFAULT_PORTS; j++) {
            if(udpDefPorts[j].port_low != 0)
                addDefaultPort(ndpi_str, &udpDefPorts[j], &ndpi_str->proto_defaults[protoId], 0, &ndpi_str
                    ->udpRoot);

            if (tcpDefPorts[j].port_low != 0)
                addDefaultPort(ndpi_str, &tcpDefPorts[j], &ndpi_str->proto_defaults[protoId], 0, &ndpi_str
                    ->tcpRoot);

            ndpi_str->proto_defaults[protoId].tcp_default_ports[j] = tcpDefPorts[j].port_low;
            ndpi_str->proto_defaults[protoId].udp_default_ports[j] = udpDefPorts[j].port_low;
        }
    }
```

After setting the basic value to the struct **proto_defaults**. The second part adds corresponding node to the binary sort tree, with the UDP port or TCP port or both as an index, through the function **addDefaultPort**.

```
    static void addDefaultPort(struct ndpi_detection_module_struct *ndpi_str, ndpi_port_range *range,
        ndpi_proto_defaults_t *def, u_int8_t customUserProto, ndpi_default_ports_tree_node_t **root) {

        for (port = range->port_low; port <= range->port_high; port++) {
            ndpi_default_ports_tree_node_t *node =
                (ndpi_default_ports_tree_node_t *) ndpi_malloc(sizeof(ndpi_default_ports_tree_node_t));
            ndpi_default_ports_tree_node_t *ret;

            node->proto = def, node->default_port = port, node->customUserProto = customUserProto;
            ret = (ndpi_default_ports_tree_node_t *) ndpi_tsearch(node, (void *) root,
                ndpi_default_ports_tree_node_t_cmp);
            if (ret != node) {
                ret->proto = def;
                ndpi_free(node);
            }
        }
    }
```

From the definition of **ndpi_build_default_ports**, it can see a protocol can support up to 5 TCP and UDP ports, store in the **ndpi_port_range** structure.

This function insert at most 5 TCP or UDP ports to the tree. For each port, using **ndpi_malloc** function to allocate a node structure, then using **ndpi_tsearch** function to insert this new node.

The return value of **ndpi_tsearch** represents a point to the node in the binary tree. If the return address is equal to the node address just allocated, means a new node is allocated in **ndpi_tsearch** function; if the return address is inequality to the node address, means an old node in the tree is found, then just replace the **proto** in this node, and free the node which just allocated.

After that, using **init_string_based_protocols** function to further initialize the **ndpi_str**.

```
    static void init_string_based_protocols (struct ndpi_detection_module_struct *ndpi_str) {
        int i;

        for (i = 0; host_match[i].string_to_match != NULL; i++)
            ndpi_init_protocol_match(ndpi_str, &host_match[i]);

        ndpi_enable_loaded_categories(ndpi_str);

        for (i = 0; ndpi_en_bigrams[i] != NULL; i++)
            ndpi_string_to_automa(ndpi_str, &ndpi_str->bigrams_automa, (char *) ndpi_en_bigrams[i], 1, 1,
                1, 0);

        for (i = 0; ndpi_en_impossible_bigrams[i] != NULL; i++)
            ndpi_string_to_automa(ndpi_str, &ndpi_str->impossible_bigrams_automa, (char *)
                ndpi_en_impossible_bigrams[i], 1, 1, 1, 0);
    }
```

First, **ndpi_init_protocol_match** function adds the URL in **host_match** array to ac trie. Then **ndpi_enable_loaded_categories** function which will be analyzed in section 4.1.3 is used to generates category tree. After that, build the other two ac trie **bigrams_automa** and **impossible_bigrams_automa**, both relates to TOR network, which will be introduce in section 7.

4.1.2 Hostname Initialization

The secondary initialization is similar to the default protocol initialization, which add the relationship between port and protocol ID to the binary sort tree. Besides, it add url and protocol ID relationship as defined in **host_match**, which act as a supplement for the port number, defined as below:

```
static ndpi_protocol_match host_match[] = {
    { "s3.ll.dash.row.aiv-cdn.net",           "AmazonVideo",
        NDPI_PROTOCOL_AMAZON_VIDEO, NDPI_PROTOCOL_CATEGORY_VIDEO,
        NDPI_PROTOCOL_FUN },
    { "s3-dub.cf.dash.row.aiv-cdn.net",  "AmazonVideo",    NDPI_PROTOCOL_AMAZON_VIDEO,
        NDPI_PROTOCOL_CATEGORY_VIDEO, NDPI_PROTOCOL_FUN },
    { "dmqdd6hw24ucf.cloudfront.net",          "AmazonVideo",
        NDPI_PROTOCOL_AMAZON_VIDEO, NDPI_PROTOCOL_CATEGORY_VIDEO,
        NDPI_PROTOCOL_FUN },

    ...
};
```

Take the first item in **host_match** array as an example. **s3.ll.dash.row.aiv-cdn.net** is the URL, and **NDPI_PROTOCOL_AMAZON_VIDEO** represents the protocol ID. Each item is added through **ndpi_init_protocol_match** function, defined as:

```
void ndpi_init_protocol_match(struct ndpi_detection_module_struct *ndpi_str, ndpi_protocol_match *
    match) {
    u_int16_t no_master[2] = {NDPI_PROTOCOL_NO_MASTER_PROTO,
        NDPI_PROTOCOL_NO_MASTER_PROTO};
    ndpi_port_range ports_a[MAX_DEFAULT_PORTS], ports_b[MAX_DEFAULT_PORTS];

    if (ndpi_str->proto_defaults[match->protocol_id].protoName == NULL) {
        ndpi_str->proto_defaults[match->protocol_id].protoName = ndpi_strdup(match->proto_name);
        ndpi_str->proto_defaults[match->protocol_id].protoId = match->protocol_id;
        ndpi_str->proto_defaults[match->protocol_id].protoCategory = match->protocol_category;
        ndpi_str->proto_defaults[match->protocol_id].protoBreed = match->protocol_breed;

        ndpi_set_proto_defaults (ndpi_str, ndpi_str->proto_defaults[match->protocol_id].protoBreed,
                    ndpi_str->proto_defaults[match->protocol_id].protoId, 0,
                    no_master, no_master, ndpi_str->proto_defaults[match->protocol_id].protoName
                    ,
                    ndpi_str->proto_defaults[match->protocol_id].protoCategory,
                    ndpi_build_default_ports (ports_a, 0, 0, 0, 0, 0) /* TCP */,
                    ndpi_build_default_ports (ports_b, 0, 0, 0, 0, 0) /* UDP */);
    }

    ndpi_add_host_url_subprotocol(ndpi_str, match->string_to_match, match->protocol_id,
        match->protocol_category, match->protocol_breed);
}
```

There are two steps of this function.

First, if the **proto_defaults** is already set by **ndpi_init_protocol_defaults** function, nothing will be changed. Otherwise, set related information in the **proto_defaults** array. Since **ports_a** and **ports_b** both are zero, so it does not change any relation between port and protocol ID, which is set in **ndpi_init_protocol_defaults** function.

The second steps will add each url to the ac trie, as below:

```
static int ndpi_add_host_url_subprotocol(struct ndpi_detection_module_struct *ndpi_str, char *_value,
    int protocol_id,
                                   ndpi_protocol_category_t category, ndpi_protocol_breed_t breed) {
    ...
    ndpi_string_to_automa(ndpi_str, &ndpi_str->host_automa, value, protocol_id, category, breed, 1);

    return(rv);
}
```

This function adds the string **value** and **protocol_id** to the ac trie **host_automa**. The function **ndpi_string_to_automa** will be introduced in section 3.3.4.

4.1.3 Category Initialization

Traditionally nDPI was used by ntopng to detect flows L7 protocol. With the advent of more and more protocols, speaking about single protocols is often too difficult. Users usually are not interested in the specific protocol but rather on a whole group of protocols. For example, its easier to reason about VPN traffic as a whole rather than a particular VPN implementation.

For these reasons, nDPI (and ntopng) has been extended to provide a logical grouping of protocols, called Categories. With Categories its possible, for example, to get an idea of the network traffic of a host.

nDPI has defined some type of categories as below:

```
typedef enum {
    NDPI_PROTOCOL_CATEGORY_UNSPECIFIED = 0, /* For general services and unknown
        protocols */
    NDPI_PROTOCOL_CATEGORY_MEDIA, /* Multimedia and streaming */
    NDPI_PROTOCOL_CATEGORY_VPN,    /* Virtual Private Networks */

    ...

    NDPI_PROTOCOL_NUM_CATEGORIES
} ndpi_protocol_category_t;
```

User can define their own category, specify by **NDPI_PROTOCOL_CATEGORY_CUSTOM_1**, etc.

Take a look at the **ndpi_enable_loaded_categories** function:

Listing 38: 1st Part of **ndpi_enable_loaded_categories**

```
int ndpi_enable_loaded_categories(struct ndpi_detection_module_struct *ndpi_str) {
    ...

    for (i = 0; category_match[i].string_to_match != NULL; i++)
        ndpi_load_category(ndpi_str, category_match[i].string_to_match, category_match[i].
            protocol_category);
```

This function Traversal **category_match** array, using **ndpi_load_category** to save them in the **custom_categories**. Note that, **category_match** contains both IP address and URL corresponding to a category, as below:

```
static ndpi_category_match category_match[] = {
    { ".edgecastcdn.net",                NDPI_PROTOCOL_CATEGORY_MEDIA },
    { ".hwcdn.net",                      NDPI_PROTOCOL_CATEGORY_MEDIA },

    ...

    { "8.28.124.0/24",                  NDPI_PROTOCOL_CATEGORY_STREAMING },
    { "8.28.125.0/24",                  NDPI_PROTOCOL_CATEGORY_STREAMING },

    ...

    { NULL, 0 }
};
```

It can be seen that both URL and IP address exist in this array simultaneously, **ndpi_load_category** will use two functions to resolve them, as below:

```
int ndpi_load_category(struct ndpi_detection_module_struct *ndpi_struct, const char *ip_or_name,
                       ndpi_protocol_category_t category) {
    int rv;

    rv = ndpi_load_ip_category(ndpi_struct, ip_or_name, category);

    if(rv < 0) {
        rv = ndpi_load_hostname_category(ndpi_struct, ip_or_name, category);
    }

    return(rv);
}
```

Inside this function, **ndpi_load_ip_category** is used to add ip address to patricia tree **ipAddresses_shadow**, and **ndpi_load_hostname_category** treat **ip_or_name** as URL and add to the ac trie **hostnames_shadow**.

There two function is similar to the description above, the only difference is the representative stored in either patricia three or ac trie is the category instead of protocol ID.

Listing 39: 2nd Part of **ndpi_enable_loaded_categories**

```
    ac_automata_release((AC_AUTOMATA_t *) ndpi_str->custom_categories.hostnames.ac_automa,
    ac_automata_finalize((AC_AUTOMATA_t *) ndpi_str->custom_categories.hostnames_shadow.
        ac_automa);
    ndpi_str->custom_categories.hostnames.ac_automa = ndpi_str->custom_categories.
        hostnames_shadow.ac_automa;
    ndpi_str->custom_categories.hostnames_shadow.ac_automa = ac_automata_init(ac_match_handler);

    if(ndpi_str->custom_categories.ipAddresses != NULL)
        ndpi_Destroy_Patricia((patricia_tree_t *) ndpi_str->custom_categories.ipAddresses,
            free_ptree_data);

    ndpi_str->custom_categories.ipAddresses = ndpi_str->custom_categories.ipAddresses_shadow;
    ndpi_str->custom_categories.ipAddresses_shadow = ndpi_New_Patricia(32 /* IPv4 */);

    ndpi_str->custom_categories.categories_loaded = 1;

    return(0);
}
```

The second part stores points of the **"shadow"** tree, which is initialized in the first part to their corresponding variable, i.e. the **ac_automa** and **ipAddress**. I am a little confused about these operations of **"shadow"** variables, in my opinion, it is for future use.

4.2 Dissector Initialization

Back to function **setupDetection** in **ndpiReader**.

After calling the **work_flow_init** function, ndpi_set_protocol_detection_bitmask2 will be called to initialize all default dissectors.

The first part of this function is as below:

Listing 40: 1st Part of **ndpi_set_protocol_detection_bitmask2**

```
void ndpi_set_protocol_detection_bitmask2(struct ndpi_detection_module_struct *ndpi_str,
                                          const NDPI_PROTOCOL_BITMASK *dbm) {
    NDPI_PROTOCOL_BITMASK detection_bitmask_local;
    NDPI_PROTOCOL_BITMASK *detection_bitmask = &detection_bitmask_local;
    u_int32_t  a = 0;

    NDPI_BITMASK_SET(detection_bitmask_local, *dbm);
    NDPI_BITMASK_SET(ndpi_str->detection_bitmask, *dbm);

    ndpi_str->callback_buffer_size  = 0;

    init_http_dissector (ndpi_str, &a, detection_bitmask);
    ...
```

Each bit in **detection_bitmask** specifies different application protocols, take a look at the definition of NDPI_PROTOCOL_BITMASK:

```
#define NDPI_PROTOCOL_BITMASK ndpi_protocol_bitmask_struct_t

typedef struct ndpi_protocol_bitmask_struct {
    ndpi_ndpi_mask fds_bits[NDPI_NUM_FDS_BITS];
} ndpi_protocol_bitmask_struct_t;
```

The size of **fds_bits** is **NDPI_NUM_FDS_BITS** which is 16, and each **ndpi_ndpi_mask** item is 32 bit, thus nDPI support up to $32 \times 16 = 512$ protocols.

The main job of **ndpi_set_protocol_detection_bitmask2** function is to initialize each application dissector, some match patterns and callback functions will be registered.

Take HTTP dissector initialization function **init_http_dissector** as an example. The parameter **a** calculate the total number of callback function. and **detection_bitmask** represents all the 512 protocol which is set to all 1 in above function.

```
void init_http_dissector (struct ndpi_detection_module_struct *ndpi_struct, u_int32_t *id,
     NDPI_PROTOCOL_BITMASK *detection_bitmask) {
    ndpi_set_bitmask_protocol_detection("HTTP",ndpi_struct, detection_bitmask, *id,
                                        NDPI_PROTOCOL_HTTP,
                                        ndpi_search_http_tcp,
                                        NDPI_SELECTION_BITMASK_PROTOCOL_V4_V6_TCP_WITH_PAYLOAD
                                            ,
                                        SAVE_DETECTION_BITMASK_AS_UNKNOWN,
                                        ADD_TO_DETECTION_BITMASK);
    *id += 1;
}
```

The micro **NDPI_PROTOCOL_HTTP** is the protocol ID of HTTP, and **ndpi_search_http_tcp** is the corresponding callback function which will be invoked when HTTP protocol is detected.

The micro **NDPI_SELECTION_BITMASK_PROTOCOL_V4_V6_TCP_WITH_PAYLOAD** is another properties of this dissector, which will stored in **ndpi_selection_bitmask**. It means the callback function is only invoked when the packet belongs to IPv4 or IPv6 TCP packet, also has payload. Otherwise, even HTTP protocol is detected, function **ndpi_search_http_tcp** will not be called.

The second part of **init_http_dissector** function is as below:

Listing 41: 2nd Part of **ndpi_set_protocol_detection_bitmask2**

```
    ...

    for (a = 0; a < ndpi_str->callback_buffer_size; a++) {
        if((ndpi_str->callback_buffer[a].ndpi_selection_bitmask &
                    (NDPI_SELECTION_BITMASK_PROTOCOL_INT_TCP |
                        NDPI_SELECTION_BITMASK_PROTOCOL_INT_TCP_OR_UDP |
                    NDPI_SELECTION_BITMASK_PROTOCOL_COMPLETE_TRAFFIC)) != 0) {
                memcpy(&ndpi_str->callback_buffer_tcp_payload[ndpi_str->callback_buffer_size_tcp_payload],
                        &ndpi_str->callback_buffer[a], sizeof(struct ndpi_call_function_struct));
                ndpi_str->callback_buffer_size_tcp_payload++;

                if((ndpi_str->callback_buffer[a].ndpi_selection_bitmask &
                    NDPI_SELECTION_BITMASK_PROTOCOL_HAS_PAYLOAD) == 0) {
                    memcpy(&ndpi_str->callback_buffer_tcp_no_payload[ndpi_str->
                        callback_buffer_size_tcp_no_payload],
                            &ndpi_str->callback_buffer[a], sizeof(struct ndpi_call_function_struct));
                    ndpi_str->callback_buffer_size_tcp_no_payload++;
                }
            }
        }

        ...
    }
```

According to the analysis of first part, **callback_buffer_size** stores how many callback function defined in this function. The second part traversals all these callback functions, for each function, according to the **ndpi_selection_bitmask** value, classify them into different callback buffer.

For example, if the **ndpi_selection_bitmask** of a callback function indicates it belongs to TCP category, the callback function will be copy to either **callback_buffer_tcp_payload** or **callback_buffer_tcp_no_payload** buffer. We omit the others callback buffers, such sa UDP callback buffer, and neither TCP or UDP callback buffer.

Since two or more callback functions can have the same protocol ID, it will be seen in section 5, if the callback function failed to analyze one packet, the other callback functions will be selected base on the **ndpi_selection_bitmask** value.

The procedure of function **ndpi_set_protocol_detection_bitmask2** is depicts in Fig.22.

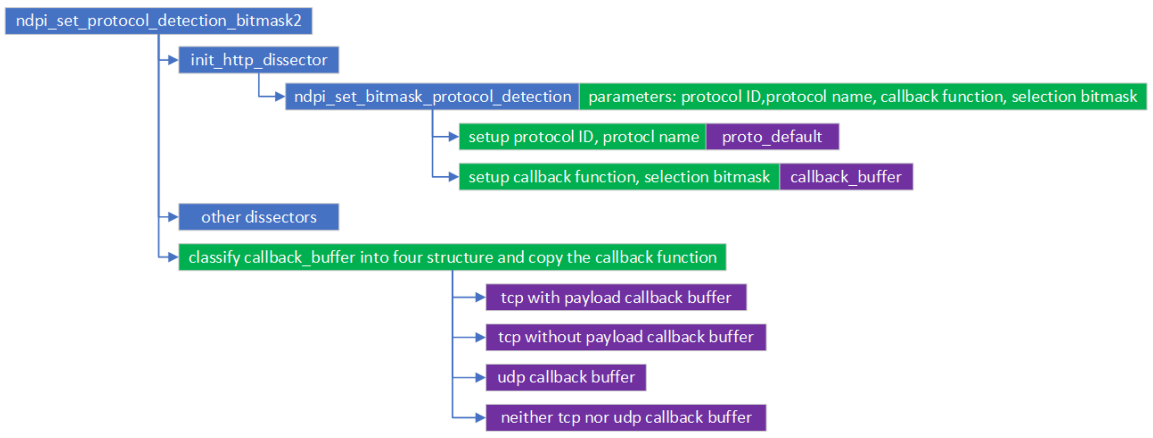

ndpi_set_protocol_detection_bitmask2
init_http_dissector
ndpi_set_bitmask_protocol_detection
parameters: protocol ID,protocol name, callback function, selection bitmask
setup protocol ID, protocl name
proto_default
setup callback function, selection bitmask
callback_buffer
other dissectors
classify callback_buffer into four structure and copy the callback function
tcp with payload callback buffer
tcp without payload callback buffer
udp callback buffer
neither tcp nor udp callback buffer

Figure 22: The procedure of **ndpi_set_protocol_detection_bitmask2**.

4.3 Load Protocol File

Back to the **setupDetection** function, if users specify **"-p"** option and assign a custom protocol files, the **ndpi_load_protocols_file** function will be called.

```
int ndpi_load_protocols_file(struct ndpi_detection_module_struct *ndpi_str, const char *path) {

    ...

    fd = fopen(path, "r");

     ...

    while ((line = fgets(line, line_len, fd)) != NULL && line[strlen(line) - 1] != '\n') {
        ndpi_handle_rule(ndpi_str, buffer, 1);
    }
}
```

This function open the assigned file, then using **ndpi_handle_rule** to handle each line:

Listing 42: 1st Part of **ndpi_handle_rule**

```
int ndpi_handle_rule(struct ndpi_detection_module_struct *ndpi_str, char *rule, u_int8_t do_add) {
     ...

    at = strrchr(rule, '@');
    at [0] = 0, proto = &at[1];

    for (i = 0, def = NULL; i < (int) ndpi_str->ndpi_num_supported_protocols; i++) {
        if(ndpi_str->proto_defaults[i].protoName && strcasecmp(ndpi_str->proto_defaults[i].
            protoName, proto) == 0) {
            def = &ndpi_str->proto_defaults[i];
            subprotocol_id = i;
            break;
        }
    }
```

For more clearly explain this function, consider two example line in file:

- host:"googlesyndacation.com"@Google
- tcp:860-865,udp:860@Wechat

ndpi_handle_rule function first checks if the line contains a special character **"@"**, then split the line into two part, the right part contains the protocol string, such as **Google** and **Wechat**, and the left part includes the information needed to match this protocol.

Then, the protocol string will compare with **protoName**, which is set by **ndpi_set_proto_defaults** function, to find if the same protocol has been registered in the ndpi detection module(**ndpi_str**).

Listing 43: 2nd Part of **ndpi_handle_rule**

```
    if(def == NULL) {
        ...
        ndpi_set_proto_defaults(
                                ndpi_str, NDPI_PROTOCOL_ACCEPTABLE, ndpi_str->
                                    ndpi_num_supported_protocols,
                                0, no_master, no_master, proto,
                                NDPI_PROTOCOL_CATEGORY_UNSPECIFIED,
                                ndpi_build_default_ports(ports_a, 0, 0, 0, 0, 0),
                                ndpi_build_default_ports(ports_b, 0, 0, 0, 0, 0));
        def = &ndpi_str->proto_defaults[ndpi_str->ndpi_num_supported_protocols];
        subprotocol_id = ndpi_str->ndpi_num_supported_protocols;
        ndpi_str->ndpi_num_supported_protocols++, ndpi_str->ndpi_num_custom_protocols++;
    }
```

According to the discussion of first part, **bf** is **NULL** means the protocol does not register previously, thus using **ndpi_set_proto_defaults** function to register an item in the **proto_defaults** array.

Listing 44: 3rd Part of **ndpi_handle_rule**

```
    while ((elem = strsep(&rule, ",")) != NULL) {
        if(strncmp(attr, "tcp:", 4) == 0)
            is_tcp = 1, value = &attr[4];
        else if(strncmp(attr, "udp:", 4) == 0)
            is_udp = 1, value = &attr[4];
        else if(strncmp(attr, "ip:", 3) == 0)
            is_ip = 1, value = &attr[3];
        else if(strncmp(attr, "host:", 5) == 0) {
            ...
            for(i=0; i<max_len; i++) value[i] = tolower(value[i]);
        }
```

In the left side of character **"@"**, each match pattern is separated by character **","**, the pattern can be a TCP port, or UDP port, or IP address, or host URL.

Listing 45: 4th Part of **ndpi_handle_rule**

```
    if(is_tcp || is_udp) {
        u_int p_low, p_high;

        if(sscanf(value, "%u-%u", &p_low, &p_high) == 2)
            range.port_low = p_low, range.port_high = p_high;
        else
                range.port_low = range.port_high = atoi(&elem[4]);

        if(do_add)
            addDefaultPort(ndpi_str, &range, def, 1,
                is_tcp ? &ndpi_str->tcpRoot : &ndpi_str->udpRoot, __FUNCTION__, __LINE__);
        else
            removeDefaultPort(&range, def, is_tcp ? &ndpi_str->tcpRoot : &ndpi_str->udpRoot);
    }
```

If the left part of line is UDP port or TCP port, first check if it is a range with character **"-"**, then extract the low port and high port.

do_add flags is always 1 pass by **ndpi_load_protocols_file** function, so function **addDefaultPort** is called to add this port number to corresponding binary sort tree.

Listing 46: 5th Part of **ndpi_handle_rule**

```
        else if(is_ip) {
            ndpi_add_host_ip_subprotocol(ndpi_str, value, subprotocol_id);
        } else {
            if(do_add)
                ndpi_add_host_url_subprotocol(ndpi_str, value, subprotocol_id,
                    NDPI_PROTOCOL_CATEGORY_UNSPECIFIED,
                                        NDPI_PROTOCOL_ACCEPTABLE);
            else
                ndpi_remove_host_url_subprotocol(ndpi_str, value, subprotocol_id);
        }
    }

    return(0);
}
```

In the last part of **ndpi_load_protocols_file** function. If the left part is IP address, then add this IP address to the patricia tree through **ndpi_add_host_ip_subprotocol** function.

If the left part is host URL, add this URL to the ac trie through **ndpi_add_host_url_subprotocol** function.

Both function is as below:

```
static int ndpi_add_host_ip_subprotocol(struct ndpi_detection_module_struct *ndpi_str,
                                        char *value, u_int16_t protocol_id) {
    ...
    add_to_ptree(ndpi_str->protocols_ptree, AF_INET, &pin, bits)
    ...
}

static int ndpi_add_host_url_subprotocol(struct ndpi_detection_module_struct *ndpi_str, char *_value,
    int protocol_id,
                                        ndpi_protocol_category_t category, ndpi_protocol_breed_t breed) {
    ...
    ndpi_string_to_automa(ndpi_str, &ndpi_str->host_automa, value, protocol_id, category, breed, 1);
    ...
}
```

Inside both function, **add_to_ptree** and **ndpi_string_to_automa** have be introduced in section 3.2.3 and section 3.3.4, respectively.

Finally, the **ndpi_load_protocols_file** function can be denoted by Fig.23.

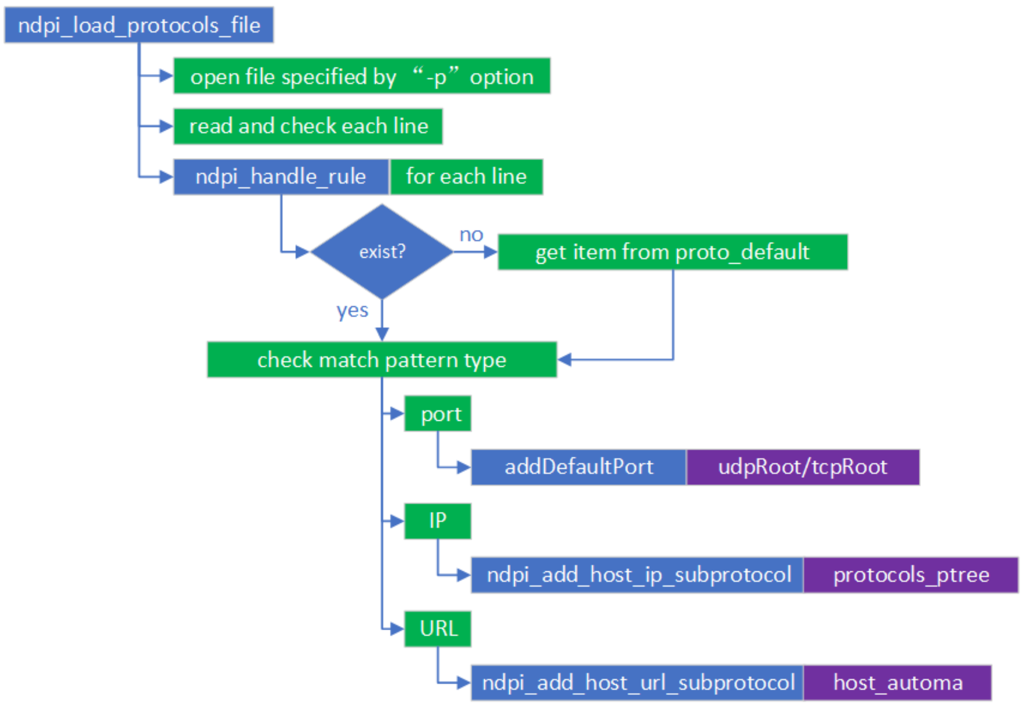


Figure 23: The procedure of **ndpi_load_protocols_file**.

4.4 Other Initialization

Some secondary initialization functions of nDPI will be introduced in this section.

4.4.1 Load Category File

Similar to section 4.3, if users specify **"-c"** option and assign a custom category file, function **ndpi_load_categories_file** will be called, as below:

```
int  ndpi_load_categories_file (struct ndpi_detection_module_struct *ndpi_str, const char *path) {
    ...
    fd = fopen(path, "r");

    while (1) {
        line = fgets(buffer, sizeof(buffer), fd);
        ...
        ndpi_load_category(ndpi_str, name, (ndpi_protocol_category_t) atoi(category));
    }
}
```

This function open the custom category files, read each line, extract the IP address or URL, then using **ndpi_load_category** to add them to the patricia tree or ac trie.

4.4.2 AC Trie Finalize

The last part of initialization is to finalize the ac trie, as below:

```
void  ndpi_finalize_initalization (struct ndpi_detection_module_struct *ndpi_str) {
    u_int i;

    for (i = 0; i < 4; i++) {
        ndpi_automa *automa;

        switch(i) {
        case 0:
            automa = &ndpi_str->host_automa;
            break;
        ...
        }

        if(automa) {
            ac_automata_finalize((AC_AUTOMATA_t *) automa->ac_automa);
            automa->ac_automa_finalized = 1;
        }
    }
}
```

It can be seen that this function will call **ac_automata_finalize** to finish constructing a ac trie. Four ac trie will be check, the **host_automa**, **content_automa**, **bigrams_automa** and **impossible_bigrams_automa**.

More detail about ac trie and this function will be introduced in section 3.3.

4.5 Summary

All the procedure of nDPI initialization is summarized in this section. **setupDetection** is the entry of ndpi initialization in **ndpiReader**, depicts in Fig.24.

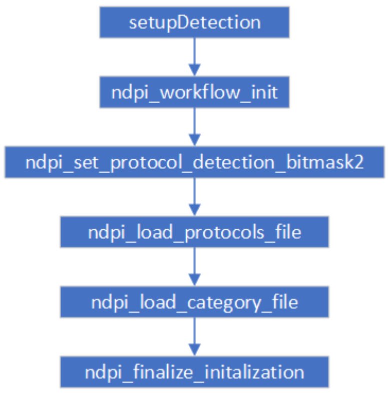


Figure 24: The main function in **setupDetection**.

Where **ndpi_workflow_init** constructs the relation between protocol ID and port, IP address, URL. **ndpi_set_protocol_detection_bitmask2** is responsible to connect each protocol ID to each own dissector. After that, using **ndpi_load_protocols_file** and **ndpi_load_categories_file** to load and resolve custom files. Finally using **ndpi_finalize_initialization** function to finish the building process of ac trie.

5 nDPI Process

A packet processing example in **ndpiReader** is shown in section 2.3.2. In this section, some key ndpi lib functions relate to this processing are present. Section 5.1 will introduce how match a flow structure given a packet. And the main processing procedure is described in section 5.2. A summary is given in section 5.3.

5.1 Search Flow

For each packet, the first step of **packet_processing** function is to match and find the flow information through **get_ndpi_flow_info** function. Each packet must belong to one flow. The flow may be created base on this packet or has been created based on previous packets. **get_ndpi_flow_info** function is as below:

Listing 47: 1st Part of **get_ndpi_flow_info**

```
static struct ndpi_flow_info *get_ndpi_flow_info (...) {
    ...
    l4_offset  = iph->ihl * 4;
    l3 = (const u_int8_t*)iph;
    *proto = iph->protocol;
    l4 =& ((const u_int8_t *) l3)[ l4_offset ];

    if(*proto == IPPROTO_TCP && l4_packet_len >= sizeof(struct ndpi_tcphdr)) {
        *tcph = (struct ndpi_tcphdr *)l4;
        *sport = ntohs((*tcph)->source), *dport = ntohs((*tcph)->dest);
        tcp_len = ndpi_min(4*(*tcph)->doff, l4_packet_len);
        *payload = (u_int8_t*)&l4[tcp_len];
        *payload_len = ndpi_max(0, l4_packet_len-4*(*tcph)->doff);
        l4_data_len = l4_packet_len - sizeof(struct ndpi_tcphdr);
    } else if(*proto == IPPROTO_UDP && l4_packet_len >= sizeof(struct ndpi_udphdr)) {
        ...
    }

    ...
```

The first part of **get_ndpi_flow_info** function extracts the level 3 and level 4 header. After checking the IP protocol field in level 3 header, the level 4 protocol can be determined.

Then, if it is a TCP header, some useful information will be extracted too, such as source port, destination port, the payload pointer and length, etc.

Note that, **payload_len** and **l4_data_len** may not be equal, since there are some option items between the header and payload.

Listing 48: 2nd Part of **get_ndpi_flow_info**

```
flow.protocol = iph->protocol, flow.vlan_id = vlan_id;
flow.src_ip = iph->saddr, flow.dst_ip = iph->daddr;
flow.src_port = htons(*sport), flow.dst_port = htons(*dport);
flow.hashval = hashval = flow.protocol + flow.vlan_id + flow.src_ip + flow.dst_ip + flow.src_port
    + flow.dst_port;

idx = hashval % workflow->prefs.num_roots;
ret = ndpi_tfind(&flow, &workflow->ndpi_flows_root[idx], ndpi_workflow_node_cmp);
```

The second part of **get_ndpi_flow_info** set the flow structure base on current packet. A hash value is calculated and used to specify one item in the **ndpi_flows_root** array. Then, since **ndpi_flows_root** is also a binary sort tree, **ndpi_tfind** can be used to find the flow from corresponding root node, which is introduced in section 3.1.2.

Note that, different with the binary sort tree introduced in function **ndpi_set_proto_defaults**, which choose the child base on the comparing result of port number, the flow binary sort tree adopts the comparing result of **ndpi_workflow_node_cmp** function when choosing the child nodes.

Listing 49: 3rd Part of **get_ndpi_flow_info**

```
    int is_changed = 0;
    if(ret == NULL) {
        u_int32_t orig_src_ip = flow.src_ip;
        u_int16_t orig_src_port = flow.src_port;
        u_int32_t orig_dst_ip = flow.dst_ip;
        u_int16_t orig_dst_port = flow.dst_port;

        flow.src_ip = orig_dst_ip;
        flow.src_port = orig_dst_port;
        flow.dst_ip = orig_src_ip;
        flow.dst_port = orig_src_port;

        is_changed = 1;

        ret = ndpi_tfind(&flow, &workflow->ndpi_flows_root[idx], ndpi_workflow_node_cmp);
    }
```

If **ndpi_tfind** function in second part doesn't find the flow node, in this part, the source port, IP address and destination port, IP address will be exchanged, flag **is_changed** indicates if the exchange operation has been done.

After change the source and destination information, call **ndpi_tfind** again to find if the flow node exists on **ndpi_flows_root**.

Listing 50: 4th Part of **get_ndpi_flow_info**

```
    if(ret == NULL) {
        struct ndpi_flow_info *newflow = (struct ndpi_flow_info*)ndpi_malloc(sizeof(struct
            ndpi_flow_info));

        memset(newflow, 0, sizeof(struct ndpi_flow_info));
        newflow->flow_id = flow_id++;
        newflow->protocol = iph->protocol, newflow->vlan_id = vlan_id;
        newflow->src_ip = iph->saddr, newflow->dst_ip = iph->daddr;
        newflow->src_port = htons(*sport), newflow->dst_port = htons(*dport);
         ...
```

After exchanging the source and destination information, if still can't find the node in **ndpi_flows_root**, this part will create one **ndpi_flow_info** structure, fill it with information, then insert it into the **ndpi_flows_root**.

Among them, the **flow_id** defined as a global variable, it increases when detect a new flow.

Listing 51: 5th Part of **get_ndpi_flow_info**

```
            ndpi_init_bin(&newflow->payload_len_bin, ndpi_bin_family8, PLEN_NUM_BINS);

            inet_ntop(AF_INET, &newflow->src_ip, newflow->src_name, sizeof(newflow->src_name));
            inet_ntop(AF_INET, &newflow->dst_ip, newflow->dst_name, sizeof(newflow->dst_name));

            ndpi_tsearch(newflow, &workflow->ndpi_flows_root[idx], ndpi_workflow_node_cmp);

             ...

            return newflow;
        }
    }
```

The 5th part convert the IP address from binary to string and stores in **src_name** and **dst_name**, then insert the created **newflow** into the binary sort tree **ndpi_flows_root**. Part of the code relates to entropy calculation is omit, since they only influent printing and DDOS calculation.

Listing 52: 6th Part of **get_ndpi_flow_info**

```
        else {
            struct ndpi_flow_info *rflow = *(struct ndpi_flow_info**)ret;

            if(is_changed) {
                if(rflow->src_ip == iph->saddr && rflow->dst_ip == iph->daddr && rflow->src_port ==
                     htons(*sport)
                   && rflow->dst_port == htons(*dport))
                   *src = rflow->dst_id, *dst = rflow->src_id, * src_to_dst_direction = 0, rflow->
                        bidirectional = 1;
                else
                   *src = rflow->src_id, *dst = rflow->dst_id, * src_to_dst_direction = 1;
            } else {
                ...
            }
            ...
            return(rflow);
        }
    }
```

When enter this else statement, the flow already exists on **ndpi_flows_root**. The flag **is_changed** indicates if the **rflow** has exchanged the source and destination information, then check if the flow's source and destination information equals to the current packet(**iph**), base on this result, set the direction and corresponding value, finally return the flow.

5.2 Process Packet

As introduced in section 2.3.2, **ndpi_detection_process_packet** will be called by ndpiReader to resolve a packet. The next section 5.2.1 will introduce the main routine of processing a packet, and more details will be presented in other sub-sections.

5.2.1 Main Procedure

ndpi_detection_process_packet is split into several parts, as below:

Listing 53: 1st Part of **ndpi_detection_process_packet**

```
    ndpi_protocol ndpi_detection_process_packet (...) {

        ret.category = flow->category;
        ret.master_protocol = flow->detected_protocol_stack[1],
        ret.app_protocol = flow->detected_protocol_stack[0];

        if(flow->detected_protocol_stack[0] != NDPI_PROTOCOL_UNKNOWN) {
            if(flow->check_extra_packets) {
                ndpi_process_extra_packet(ndpi_str, flow, packet, packetlen, current_time_ms, src, dst);
                ret.master_protocol = flow->detected_protocol_stack[1],
                ret.app_protocol = flow->detected_protocol_stack[0],
                ret.category = flow->category;
                goto invalidate_ptr;
            } else
                goto ret_protocols;
        }
```

detected_protocol_stack records the history detection results, if exists, means the previous guessed protocol relates to this flow has been confirmed and stored in this structure. If the protocol is not confirmed yet, goto next part of this function, otherwise, check **check_extra_packets** flags to see if this flow still need to process the new packet, for example, for a HTTP flow, it need to handle at most 5 extra packets after confirming the protocol.

check_extra_packets is set by each protocol dissector, and may be clear in **ndpi_process_extra_packet** function. If set, **ndpi_process_extra_packet** will be called to update the detection results, otherwise, goto **ret_protocols**.

In summary, **ndpi_process_extra_packet** offers a mechanism for each protocol, that even the protocol is guessed and confirmed previously, this flow still has chance to change the result base on later packets.

ndpi_process_extra_packet function is as below:

Listing 54: 1st Part of **ndpi_process_extra_packet**

```
void ndpi_process_extra_packet(struct ndpi_detection_module_struct *ndpi_str, struct ndpi_flow_struct
    *flow, const unsigned char *packet, const unsigned short packetlen, const u_int64_t
    current_time_ms, struct ndpi_id_struct *src, struct ndpi_id_struct *dst) {
  flow->packet.iph = (struct ndpi_iphdr *) packet;
  ndpi_init_packet_header(ndpi_str, flow, packetlen);
  ndpi_connection_tracking(ndpi_str, flow);
```

The first part extract level 4 information by using **ndpi_init_packet_header** function, which is introduced in section 5.2.2. Then **ndpi_connection_tracking** will detect the TCP states, an determine if there exist a TCP retransmission, more detail about this function is in section 5.2.3.

Listing 55: 2nd Part of **ndpi_process_extra_packet**

```
  if(flow->extra_packets_func) {
    if((flow->extra_packets_func(ndpi_str, flow)) == 0)
      flow->check_extra_packets = 0;

    if(++flow->num_extra_packets_checked == flow->max_extra_packets_to_check)
      flow->extra_packets_func = NULL;
  }
}
```

extra_packets_func is default **NULL**, and maybe set by some dissectors, for example, **extra_packets_func** in HTTP is **ndpi_search_http_tcp_again**.

If **extra_packets_func** return 0, means the previous confirmed protocol doesn't need to check extra packets and change the guessed results, thus further packets detection can be stop in this flow. In this case, set **check_extra_packets** to **NULL** to discard further extra packets processing. Otherwise, the detection should be continue.

Also, if **num_extra_packets_checked** is larger than the predefined **max_extra_packets_to_check**, **extra_packets_func** is clear, which indicates this flow has lose the chance to modify the detection result. Note that **max_extra_packets_to_check** is also set by each dissector, for HTTP dissector, it is 5.

Back to the **ndpi_detection_process_packet** function, the second part is as:

Listing 56: 2nd Part of **ndpi_detection_process_packet**

```
  ...
  ndpi_init_packet_header(ndpi_str, flow, packetlen);
  flow->src = src, flow->dst = dst;
  ndpi_connection_tracking(ndpi_str, flow);
```

The second part is similar to the **ndpi_process_extra_packet** function. **ndpi_init_packet_header** is used to extract the level 4 information, and **ndpi_connection_tracking** tracks the TCP connection state.

Listing 57: 3nd Part of **ndpi_detection_process_packet**

```

  ndpi_selection_packet = NDPI_SELECTION_BITMASK_PROTOCOL_COMPLETE_TRAFFIC;
  if(flow->packet.iph != NULL)
```

```
        ndpi_selection_packet  |= NDPI_SELECTION_BITMASK_PROTOCOL_IP |
            NDPI_SELECTION_BITMASK_PROTOCOL_IPV4_OR_IPV6;

    if (flow->packet.tcp != NULL)
        ndpi_selection_packet  |= (NDPI_SELECTION_BITMASK_PROTOCOL_INT_TCP |
            NDPI_SELECTION_BITMASK_PROTOCOL_INT_TCP_OR_UDP);

    ...
```

The **ndpi_selection_packet** which represents type of this packet is determined in this part. Five basic element will combine to form this value, they are IP, TCP, UDP, payload and TCP retransmission. These elements are calculated in either **ndpi_init_packet_header** or **ndpi_connection_tracking** function.

Listing 58: 4th Part of **ndpi_detection_process_packet**

```
    if ((! flow->protocol_id_already_guessed) && (flow->packet.iph)) {

        flow->protocol_id_already_guessed = 1;

        ...

        flow->guessed_protocol_id = (int16_t) ndpi_guess_protocol_id(ndpi_str, flow, protocol, sport,
            dport, &user_defined_proto);
        flow->guessed_host_protocol_id = ndpi_guess_host_protocol_id(ndpi_str, flow);
```

In 4th part, the if statement decides if **guessed_protocol_id** and **guessed_host_protocol_id** have been guessed, for a new flow just created, **protocol_id_already_guessed** is 0.

If the flow is just created, 2 main guessed functions are used to guess the protocol ID. **ndpi_guess_protocol_id** guess the protocol ID by TCP or UDP port through the binary sort tree **tcpRoot** or **udpRoot**; **ndpi_guess_host_protocol_id** guess the protocol ID by IP address through the patricia tree **protocols_ptree**. They are introduced in section 5.2.4 and section 5.2.5, respectively.

Listing 59: 5st Part of **ndpi_detection_process_packet**

```
        if (flow->guessed_protocol_id >= NDPI_MAX_SUPPORTED_PROTOCOLS) {
            ret.master_protocol = NDPI_PROTOCOL_UNKNOWN,
            ret.app_protocol = flow->guessed_protocol_id ? flow->guessed_protocol_id : flow->
                guessed_host_protocol_id;
             ndpi_fill_protocol_category (ndpi_str, flow, &ret);
            goto invalidate_ptr;
        }
```

If **guessed_protocol_id** is larger than **NDPI_MAX_SUPPORTED_PROTOCOLS**, means the protocol ID is customized. In the perspective of nDPI, all the customized protocols are the application protocol, not the real protocols, thus **master_protocol** will be set to **NDPI_PROTOCOL_UNKNOWN**, and **app_protocol** is set to either **guessed_protocol_id** or **guessed_host_protocol_id**.

Then, category will be fill in this **ret** by **ndpi_fill_protocol_category** function, which will be introduced in section 5.2.6. Since the protocol ID is customized, there must be no corresponding dissector to analyze the packet, so finally goto the **invalidate_ptr** and return.

Listing 60: 6st Part of **ndpi_detection_process_packet**

```
        if (user_defined_proto && flow->guessed_protocol_id != NDPI_PROTOCOL_UNKNOWN) {
            if (flow->packet.iph) {
                if (flow->guessed_host_protocol_id != NDPI_PROTOCOL_UNKNOWN) {
                    u_int8_t protocol_was_guessed;
                    ret = ndpi_detection_giveup(ndpi_str, flow, 0, &protocol_was_guessed);
                }
                 ndpi_fill_protocol_category (ndpi_str, flow, &ret);
```

```
                    goto invalidate_ptr;
                }
            }
```

user_defined_proto specify if the protocol is detected according to a customized properties in a protocol. According to previous analysis, when going into this if statement, it means the protocol ID is not customized, such as HTTP, SKYPE, etc. But the information relates to this protocol is customized, such as the port number, or the IP address.

For example, if user add TCP port 3000 to HTTP protocol through customized protocol file, which is introduced in section 4.3. Then, if a packet with TCP port 80 arrive, the **user_defined_proto** will be 0, if the TCP port is 3000, the **user_defined_proto** will be set.

When **user_defined_proto** is 1, if this is a IP packet, and the protocol ID is successfully guessed by the IP address, then **ndpi_detection_giveup** is used to cancel this detection. Note that the third parameter **enable_guess** is **0**, which means this function will use the history detection result to replace the current detection result. More details will be introduced in section 5.2.8.

Listing 61: 6st Part of **ndpi_detection_process_packet**

```
            else {
                if(flow->packet.iph) {
                    flow->guessed_host_protocol_id = ndpi_guess_host_protocol_id(ndpi_str, flow);
                }
            }
        }
```

This part of code re-guess the **guessed_host_protocol_id**, **I think this code is redundant** and can be deleted.

Listing 62: 7st Part of **ndpi_detection_process_packet**

```
        if(flow->guessed_host_protocol_id >= NDPI_MAX_SUPPORTED_PROTOCOLS) {
            ret.master_protocol = flow->guessed_protocol_id;
            ret.app_protocol = flow->guessed_host_protocol_id;
            num_calls = ndpi_check_flow_func(ndpi_str, flow, &ndpi_selection_packet);
            ndpi_fill_protocol_category(ndpi_str, flow, &ret);
            goto invalidate_ptr;
        }
```

When enter this **if** statement, means the **guessed_protocol_id** is not customized protocol or **NULL**. Then, if the **guessed_host_protocol_id** is customized, **ndpi_check_flow_func** and **ndpi_fill_protocol_category** will be called, which is introduced in section 5.2.7 and section 5.2.6 , respectively. Then return after calling the **ndpi_fill_protocol_category** to update category.

Listing 63: 8st Part of **ndpi_detection_process_packet**

```
        num_calls = ndpi_check_flow_func(ndpi_str, flow, &ndpi_selection_packet);
```

Then, in general case, **ndpi_check_flow_func** will be called, which find a matched dissector to analyze the packet further. The return value **num_calls** indicates how many dissectors are traversal to guess and confirm the protocol. More details about the callback function are introduced in section 5.2.7.

Listing 64: 10st Part of **ndpi_detection_process_packet**

```
    ret_protocols:
        if(flow->detected_protocol_stack[1] != NDPI_PROTOCOL_UNKNOWN) {
            ret.master_protocol = flow->detected_protocol_stack[1];
            ret.app_protocol = flow->detected_protocol_stack[0];

            if(ret.app_protocol == ret.master_protocol)
                ret.master_protocol = NDPI_PROTOCOL_UNKNOWN;
        } else
            ret.app_protocol = flow->detected_protocol_stack[0];
```

When enter the **ret_protocols** tag, means all the variables relate to the **flow** and **packet** are updated. Among them, the most important variables are **detected_protocol_stack**. The remaining job is to collect information to determine the return value. Two cases is considered here:

- **detected_protocol_stack[1] is not NULL**: set the corresponding **master_protocol** and **app_protocol**, and if they are equal, clear the **master_protocol**.
- **detected_protocol_stack[1] is NULL**: only set the **app_protocol**.

Listing 65: 11st Part of **ndpi_detection_process_packet**

```
        if((flow->category == NDPI_PROTOCOL_CATEGORY_UNSPECIFIED) && (ret.app_protocol
            != NDPI_PROTOCOL_UNKNOWN))
            ndpi_fill_protocol_category(ndpi_str, flow, &ret);
        else
            ret.category = flow->category;
```

After fill the protocol, this part of code fill the category if it is undefined.

Listing 66: 12st Part of **ndpi_detection_process_packet**

```
        if((flow->num_processed_pkts == 1) && (ret.master_protocol ==
            NDPI_PROTOCOL_UNKNOWN) &&
            (ret.app_protocol == NDPI_PROTOCOL_UNKNOWN) && flow->packet.tcp && (flow->
                packet.tcp->syn == 0) &&
            (flow->guessed_protocol_id == 0)) {
            u_int8_t protocol_was_guessed;
            ret = ndpi_detection_giveup(ndpi_str, flow, 0, &protocol_was_guessed);
        }
```

The if statement represents a TCP flow, which have only one packet, and the first packet is not a SYN packet, and no protocol is detected through this packet, it means an unusual TCP flow. In this case, just cancel this detection through **ndpi_detection_giveup** function.

Listing 67: 12st Part of **ndpi_detection_process_packet**

```
        if((ret.master_protocol == NDPI_PROTOCOL_UNKNOWN) && (ret.app_protocol !=
            NDPI_PROTOCOL_UNKNOWN) &&
            (flow->guessed_host_protocol_id != NDPI_PROTOCOL_UNKNOWN)) {
            ret.master_protocol = ret.app_protocol;
            ret.app_protocol = flow->guessed_host_protocol_id;
        }

        ...
```

Since **app_protocol** is more specifically and has higher priority than **master_protocol** corresponding to the **guessed_host_protocol_id**. If the **master_protocol** is **NULL** and **app_protocol** is not **NULL**, we can set the **master_protocol** to **app_protocol**, and change **app_protocol** to **guessed_host_protocol_id**.

Some codes relates to security detection is omit.

Listing 68: 13st Part of **ndpi_detection_process_packet**

```
        if(num_calls == 0)
            flow->fail_with_unknown = 1;

    invalidate_ptr:
```

```
    flow->packet.iph = NULL, flow->packet.tcp = NULL, flow->packet.udp = NULL, flow->packet.
        payload = NULL;
     ndpi_reset_packet_line_info (&flow->packet);

    return(ret);
}
```

The last part first checks the **num_calls**, a zero means the packet enter the corresponding dissector, but not return correct result. In this case, set the **fail_with_unknown**, which will pause further packet detection in the flow.

5.2.2 Packet Header Detection

ndpi_init_packet_header will collect information about level 3 and level 4 from packet header, as below:

Listing 69: 1st Part of **ndpi_init_packet_header**

```
static int ndpi_init_packet_header(struct ndpi_detection_module_struct *ndpi_str,
                                   struct ndpi_flow_struct *flow, unsigned short packetlen) {

    flow->packet.payload_packet_len = 0;
    flow->packet.l4_packet_len = 0;
    flow->packet.l3_packet_len = packetlen;
    flow->packet.tcp = NULL, flow->packet.udp = NULL;
    flow->packet.generic_l4_ptr = NULL;

    ndpi_apply_flow_protocol_to_packet(flow, &flow->packet);
    l3len = flow->packet.l3_packet_len;
    decaps_iph = flow->packet.iph;
```

For a new received packet, when first enter this function, the only information obtained is the level 3 packet length(packetlen). **ndpi_apply_flow_protocol_to_packet** apply the history information to this new packet, as below:

```
void ndpi_apply_flow_protocol_to_packet(struct ndpi_flow_struct *flow, struct ndpi_packet_struct *
    packet) {
    memcpy(&packet->detected_protocol_stack, &flow->detected_protocol_stack, sizeof(packet->
        detected_protocol_stack));
    memcpy(&packet->protocol_stack_info, &flow->protocol_stack_info, sizeof(packet->
        protocol_stack_info));
}
```

Two important variable is copied to the packet, **detected_protocol_stack** is the history protocol detected result, and **protocol_stack_info** is for future usage.

Listing 70: 2nd Part of **ndpi_init_packet_header**

```
    l4_result = ndpi_detection_get_l4_internal(ndpi_str, (const u_int8_t *) decaps_iph, l3len, &l4ptr,
        &l4len, &l4protocol, 0);

    flow->packet.l4_protocol = l4protocol;
    flow->packet.l4_packet_len = l4len;
    flow->l4_proto = l4protocol;
```

ndpi_detection_get_l4_internal collect the information of level 4, includes **l4_protocol**, **l4len** and **l4ptr**.

```
static u_int8_t ndpi_detection_get_l4_internal (...) {
    ...
    if(iph != NULL && ndpi_iph_is_valid_and_not_fragmented(iph, l3_len)) {
        u_int16_t len = ntohs(iph->tot_len);
        u_int16_t hlen = (iph->ihl * 4);

        l4ptr = (((const u_int8_t *) iph) + iph->ihl * 4);

        if(len == 0)
            len = l3_len;

        l4len = (len > hlen) ? (len - hlen) : 0;
        l4protocol = iph->protocol;
    } else {
        return(1);
    }
    ...
    return(0);
}
```

If the IP packet is fragmented, level 4 information will not be collected. Otherwise, level 4 information will be calculated:

- **l4ptr**: the beginning position of level 4, calculated by add IP header length(**ihl**) to beginning position of IP header(**iph**).
- **l4len**: the length of level 4, calculate by minus the IP header length(**ihl**×4) to the IP total length **tot_len**.
- **l4protocol**: indicated in the IP header.

Then, the level 4 protocol will be check, we omit UDP and other protocol, only analyze TCP for simple.

Listing 71: 3rd Part of **ndpi_init_packet_header**

```
    if(l4protocol == IPPROTO_TCP && flow->packet.l4_packet_len >= 20) {
        flow->packet.tcp = (struct ndpi_tcphdr *) l4ptr;
        if(flow->packet.l4_packet_len >= flow->packet.tcp->doff * 4) {
            flow->packet.payload_packet_len = flow->packet.l4_packet_len - flow->packet.tcp->doff * 4;
            flow->packet.actual_payload_len = flow->packet.payload_packet_len;
            flow->packet.payload = ((u_int8_t *) flow->packet.tcp) + (flow->packet.tcp->doff * 4);
```

If level 4 protocol is TCP and level 4 packet length is larger than the minimum TCP length 20, records the position of TCP header(**l4ptr**).
If TCP packet length is larger than the TCP header length, means it contains payload, then records the payload point and payload length.

Listing 72: 4th Part of **ndpi_init_packet_header**

```
            if(flow->packet.tcp->syn != 0 && flow->packet.tcp->ack == 0 && flow->init_finished !=
                0 && flow->detected_protocol_stack[0] == NDPI_PROTOCOL_UNKNOWN){

                if(flow->http.url) {
                    ndpi_free(flow->http.url);
                    flow->http.url = NULL;
                }
                ...
```

If flag **syn** is set and **ack** is zero, indicates a new TCP connection, also this connection is in the first state of TCP 3-way handshake. At this point, nDPI will clear some information in this flow, and prepare for new detection. First, the **url** is cleared, and then:

Listing 73: 5th Part of **ndpi_init_packet_header**

```
                        backup = flow->num_processed_pkts;
                        backup1 = flow->guessed_protocol_id;
                        backup2 = flow->guessed_host_protocol_id;
                        memset(flow, 0, sizeof(*(flow)));

                        flow->num_processed_pkts = backup;
                        flow->guessed_protocol_id = backup1;
                        flow->guessed_host_protocol_id = backup2;
                        flow->packet.tcp = (struct ndpi_tcphdr *) l4ptr;
                    }
                }
            }
        }
```

The **guessed_protocol_id**, **guessed_host_protocol_id** and **num_processed_pkts** are the only remain part in this flow, other information relates to this flow will be clear.

Finally, the TCP header points **l4ptr** is stored.

The procedure of function can be summarized in Fig.25.

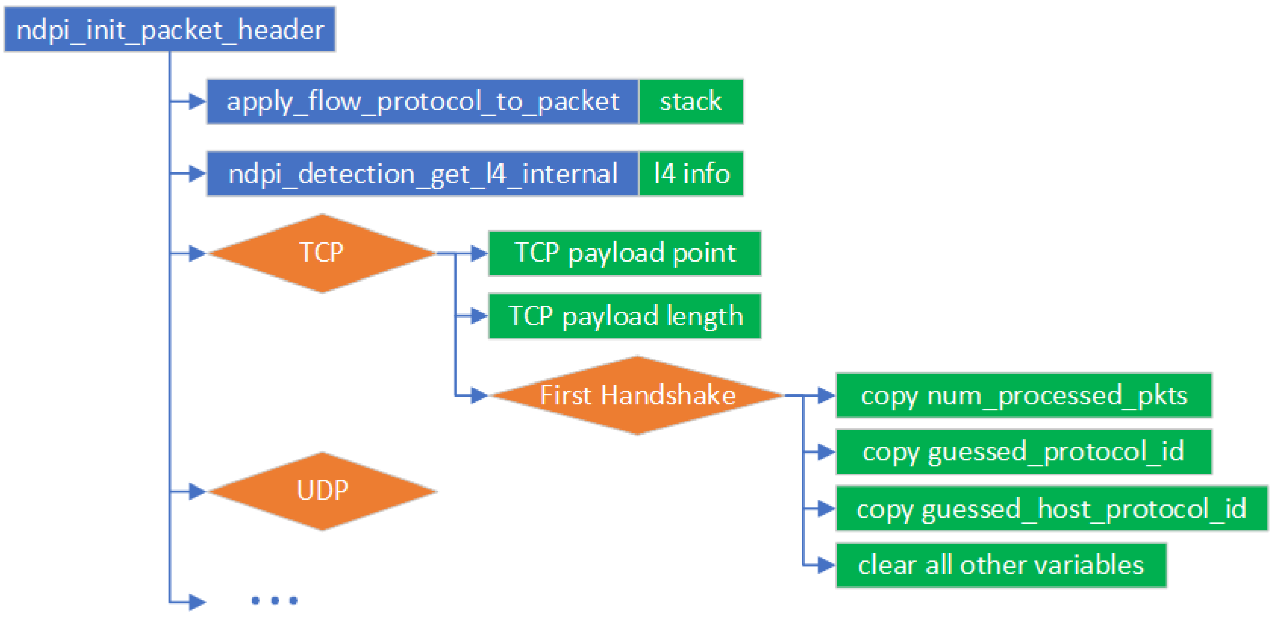


Figure 25: Procedure of **ndpi_init_packet_header**.

5.2.3 Connection Tracking

The major job of **ndpi_connection_tracking** function is to track the connection. Note that the connection is mainly for tracking the TCP connection, but this function also handles statistic of UDP and other protocols.

Listing 74: 1th Part of **ndpi_connection_tracking**

```
    void ndpi_connection_tracking(struct ndpi_detection_module_struct *ndpi_str, struct ndpi_flow_struct *
        flow) {

        packet->tcp_retransmission = 0, packet->packet_direction = 0;
        packet->packet_direction = flow->packet_direction;
        packet->packet_lines_parsed_complete = 0;

        if (flow-> init_finished  == 0) {
            flow-> init_finished  = 1;
            flow->setup_packet_direction = packet->packet_direction;
        }
```

The first part of **ndpi_connection_tracking** mainly initialize some variable, and deals with the IP level information.

Listing 75: 2nd Part of **ndpi_connection_tracking**

```
        if (tcph != NULL) {
            packet->num_retried_bytes = 0;

            if (tcph->syn != 0 && tcph->ack == 0 && flow->l4.tcp.seen_syn == 0 && flow->l4.tcp.
                seen_syn_ack == 0 && flow->l4.tcp.seen_ack == 0) {
                flow->l4.tcp.seen_syn = 1;
            }
            if (tcph->syn != 0 && tcph->ack != 0 && flow->l4.tcp.seen_syn == 1 && flow->l4.tcp.
                seen_syn_ack == 0 && flow->l4.tcp.seen_ack == 0) {
                flow->l4.tcp.seen_syn_ack = 1;
            }
            if (tcph->syn == 0 && tcph->ack == 1 && flow->l4.tcp.seen_syn == 1 && flow->l4.tcp.
                seen_syn_ack == 1 && flow->l4.tcp.seen_ack == 0) {
                flow->l4.tcp.seen_ack = 1;
            }
```

For TCP packet, consider 3 cases corresponding to different flags seen in this part:

- **syn**: indicate it is the 1st handshake, set **seen_syn**.
- **syn,ack**: this is the 2nd handshake, set **seen_syn_ack**.
- **ack**: state it is the 3rd handshake, set **seen_ack**.

seen_syn, **seen_syn_ack** and **seen_ack** represent the status in TCP connection. These information will be used in some dissector, such as SKYPE dissector.

Listing 76: 3rd Part of **ndpi_connection_tracking**

```
            if ((flow->next_tcp_seq_nr[0] == 0 && flow->next_tcp_seq_nr[1] == 0) ||
                    (flow->next_tcp_seq_nr[0] == 0 || flow->next_tcp_seq_nr[1] == 0)) {
                if (tcph->ack != 0) {
                    flow->next_tcp_seq_nr[flow->packet.packet_direction] =
                            ntohl(tcph->seq) + (tcph->syn ? 1 : packet->payload_packet_len);

                    flow->next_tcp_seq_nr[1 - flow->packet.packet_direction] = ntohl(tcph->ack_seq);
                }
            }
```

This part of code is used to record the TCP sequence number. Suppose a TCP transmission from Host A to Host B, it can be seen the next Host A to Host B sequence number should be current sequence number + 1 or the **payload_packet_len**, determined by if current transmission is for handshake or data, which indicated by the **syn** flag. And the next Host B to Host A sequence number should be **ack_seq**.

Note that nDPI is not responsible for comparing the sequence number for validate TCP connection, it acts as a router, the main purpose of tracking the sequence number is to check if a TCP retransmission happens, as below:

Listing 77: 3rd Part of **ndpi_connection_tracking**

```
            else if (packet->payload_packet_len > 0) {
                if (((u_int32_t)(ntohl(tcph->seq) - flow->next_tcp_seq_nr[packet->packet_direction])) >
                        ndpi_str->tcp_max_retransmission_window_size) {
                    packet->tcp_retransmission = 1;

                    ...
                } else {
                    ...
                }
            }

            if (tcph->rst) {
                flow->next_tcp_seq_nr[0] = 0;
                flow->next_tcp_seq_nr[1] = 0;
            }
        }
```

For TCP data transmission, if the result of current packet's sequence number minus the last recorded sequence number is larger than **tcp_max_retransmission_window_size**(default value is 65536), which indicates there must be a TCP retransmission, and **tcp_retransmission** is set.

tcp_retransmission can identify the quality of link between two host, and is one type of the **ndpi_selection_packet**, which is used to select a dissector.

Note that if **rst** flags is set, the sequence number in both direction is reset.

Listing 78: 3rd Part of **ndpi_connection_tracking**

```
        if (flow->packet_counter < MAX_PACKET_COUNTER && packet->payload_packet_len) {
            flow->packet_counter++;
        }

        if (flow->packet_direction_counter[packet->packet_direction] < MAX_PACKET_COUNTER &&
                packet->payload_packet_len) {
            flow->packet_direction_counter[packet->packet_direction]++;
        }
    }
```

In the last part, 2 counter is incremented: **packet_counter** and **packet_direction_counter**. Note that these two counter is not for print in function **printResults**, they just have impact to some dissector, for example, in HTTP dissector, if the **packet_counter** > 20, it will stop detect further packets belong to the same flow.

After all, **ndpi_connection_tracking** can be summarize in Fig.26.

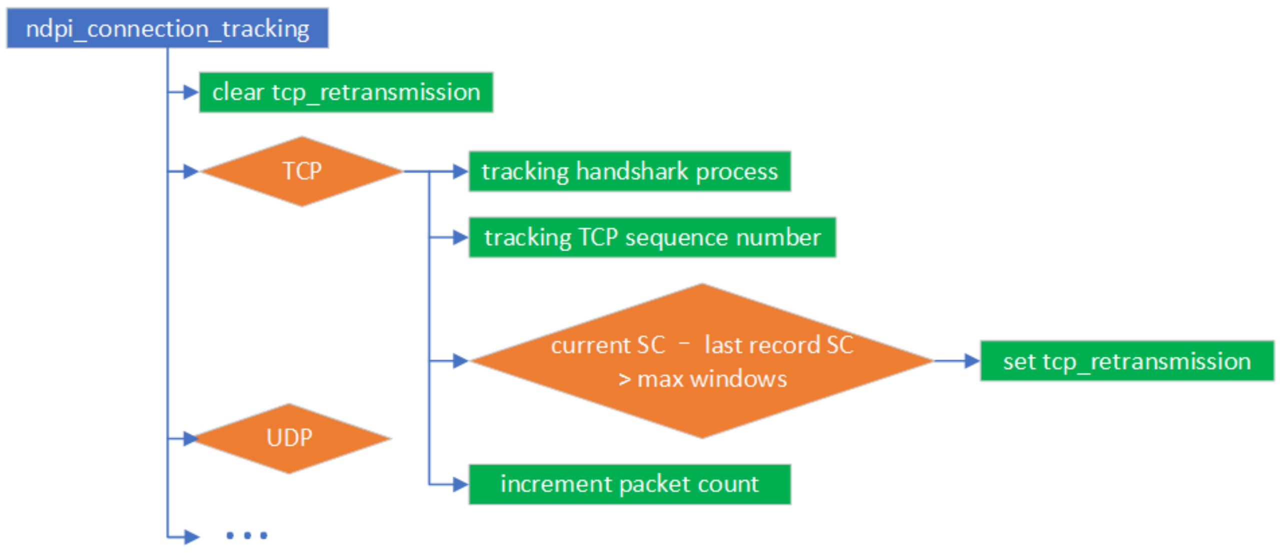


Figure 26: Procedure of **ndpi_connection_tracking**.

5.2.4 Guess Protocol By L4 Port

In function **ndpi_detection_process_packet**, the application protocol is first guessed by **ndpi_guess_protocol_id** function by using either source port and destination port. Shown as:

```
u_int16_t ndpi_guess_protocol_id(struct ndpi_detection_module_struct *ndpi_str, struct
     ndpi_flow_struct *flow, u_int8_t proto, u_int16_t sport, u_int16_t dport, u_int8_t *
     user_defined_proto) {
     *user_defined_proto = 0;

     if(sport && dport) {
          ndpi_default_ports_tree_node_t *found = ndpi_get_guessed_protocol_id(ndpi_str, proto, sport,
               dport);

          if(found != NULL) {
               if (...)
                    return(NDPI_PROTOCOL_UNKNOWN);
               else {
                    *user_defined_proto = found->customUserProto;
                    return(guessed_proto);
               }
          }
     } else {
          ...
     }

     return(NDPI_PROTOCOL_UNKNOWN);
}
```

ndpi_get_guessed_protocol_id is the function which guess the protocol by the TCP port or UDP port(both source port and destination port). If find the node, in most case it will go into the else part, which set the **user_defined_proto**. The **customUserProto** is not **NULL** if and only if user pass **"-p CustomProtocolFilePath"** to the ndpiReader, which has been analyzed in Section.4.3.

The case is omit when neither TCP or UDP is found.

```
static  ndpi_default_ports_tree_node_t  *ndpi_get_guessed_protocol_id(struct
    ndpi_detection_module_struct *ndpi_str,
        u_int8_t  proto,  u_int16_t  sport,  u_int16_t  dport) {
    ...
    if(sport  && dport) {
        int  low = ndpi_min(sport, dport);
        int  high = ndpi_max(sport, dport);

        node.default_port  = low;
        ret  = ndpi_tfind(&node, (proto == IPPROTO_TCP) ? (void *) &ndpi_str->tcpRoot : (void *)
             &ndpi_str->udpRoot,
            ndpi_default_ports_tree_node_t_cmp);

        if(ret  == NULL) {
            node.default_port  = high;
            ret  = ndpi_tfind(&node, (proto == IPPROTO_TCP) ? (void *) &ndpi_str->tcpRoot : (
                void *) &ndpi_str->udpRoot, ndpi_default_ports_tree_node_t_cmp);
        }

        if(ret)
            return(*( ndpi_default_ports_tree_node_t  **)  ret);
    }

    return(NULL);
}
```

This function lookup corresponding node which contains the protocol ID in a binary sort tree base on the port. The step can be concluded as below:

1) Base on the L4 protocol(UDP or TCP), decide searching from **tcpRoot** or **udpRoot**.

2) Using **ndpi_tfind** to search the node which match the lower port of source port and destination port.

3) If step 2) success, return result. Otherwise goto step 4).

4) Using **ndpi_tfind** to search the node which match the higher port of source port and destination port.

5) If step 4) success, return result. Otherwise return **NULL**.

UDP(udpRoot) and TCP(tcpRoot) are the binary sort trees which are constructed and introduced in section.3.1. **ndpi_tfind** is the search function, described in section.3.1.2.

In all, **ndpi_guess_protocol_id** can be concluded in Fig.27.

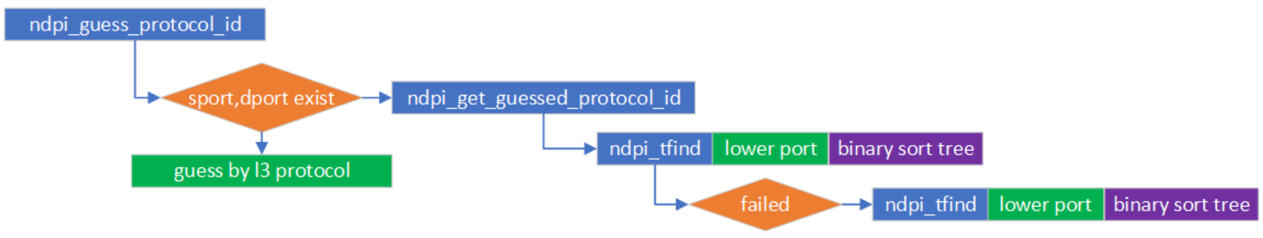


Figure 27: Procedure of **ndpi_guess_protocol_id**.

5.2.5 Guess Protocol By IP

Different from **ndpi_guess_protocol_id**, **ndpi_guess_host_protocol_id** guesses the application protocol by IP address.

Listing 79: 1st Part of **ndpi_guess_host_protocol_id**

```
u_int16_t  ndpi_guess_host_protocol_id(struct ndpi_detection_module_struct *ndpi_str, struct
    ndpi_flow_struct *flow) {

    if(flow->packet.iph) {
        struct in_addr addr;
        u_int16_t sport, dport;

        addr.s_addr = flow->packet.iph->saddr;

        if((flow->l4_proto == IPPROTO_TCP) && flow->packet.tcp)
            sport = flow->packet.tcp->source, dport = flow->packet.tcp->dest;
        else if((flow->l4_proto == IPPROTO_UDP) && flow->packet.udp)
            sport = flow->packet.udp->source, dport = flow->packet.udp->dest;
        else
            sport = dport = 0;
```

ndpi_guess_host_protocol_id first tries the source IP address, then get the source port and destination port value base on TCP or UDP protocol.

Listing 80: 2nd Part of **ndpi_guess_host_protocol_id**

```
        ret = ndpi_network_port_ptree_match(ndpi_str, &addr, sport);

        if(ret == NDPI_PROTOCOL_UNKNOWN) {
            addr.s_addr = flow->packet.iph->daddr;
            ret = ndpi_network_port_ptree_match(ndpi_str, &addr, dport);
        }
    }

    return(ret);
}
```

The second part using **ndpi_network_port_ptree_match** function to guess the protocol, and if failed, try the destination IP address. The return value **ret** represents the protocol ID.

```
u_int16_t ndpi_network_port_ptree_match(struct ndpi_detection_module_struct *ndpi_str,
                                        struct in_addr *pin,
                                        u_int16_t port) {
    ...

    node = ndpi_patricia_search_best(ndpi_str->protocols_ptree, &prefix);

    if(node) {
        if((node->value.uv.additional_user_value == 0) || (node->value.uv.additional_user_value ==
            port))
            return(node->value.uv.user_value);
    }

    return(NDPI_PROTOCOL_UNKNOWN);
}
```

The **ndpi_network_port_ptree_match** function searches the protocol ID from the patricia tree **protocols_tree**, this function have been analyzed in section 3.2.4. If successfully find the node, and if **additional_user_value** is set, then compare the port value, finally return the protocol ID in **user_value**.

additional_user_value is initialized in **ndpi_init_ptree_ipv4** with default value 0. It's value is only non-zero in TOR flow.

In summary, **ndpi_guess_host_protocol_id** can be summarized in Fig.28.

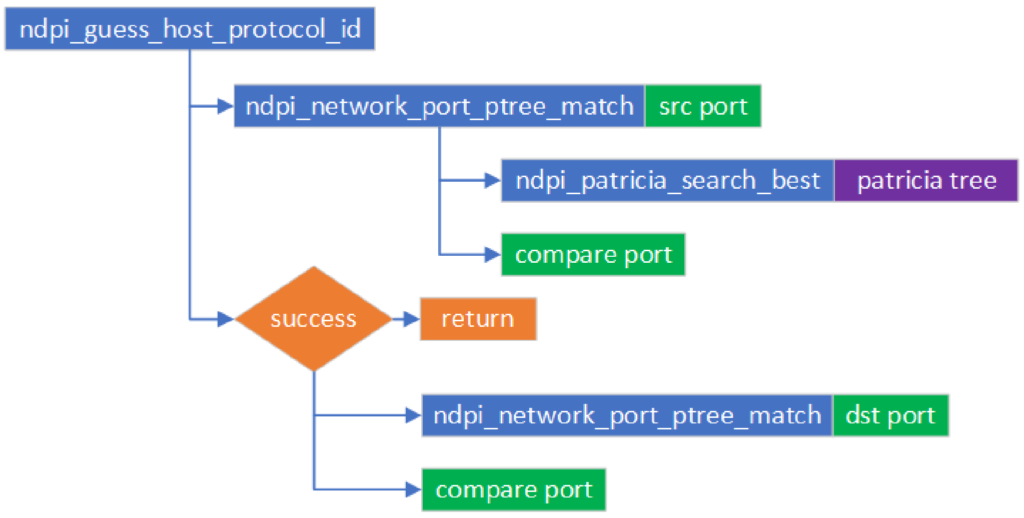

ndpi_guess_host_protocol_id
ndpi_network_port_ptree_match
src port
ndpi_patricia_search_best
patricia tree
compare port
success
return
ndpi_network_port_ptree_match
dst port
compare port

Figure 28: Procedure of **guess_host_protocol_id**.

5.2.6 Guess Category

As discussed in section 4.1.3, category is another property of a flow, which is identified by **ndpi_fill_protocol_category** function.

Listing 81: 1st Part of **ndpi_fill_protocol_category**

```
void ndpi_fill_protocol_category(struct ndpi_detection_module_struct *ndpi_str, struct
    ndpi_flow_struct *flow, ndpi_protocol *ret) {
  if(ndpi_str->custom_categories.categories_loaded) {
    if(flow->guessed_header_category != NDPI_PROTOCOL_CATEGORY_UNSPECIFIED) {
      flow->category = ret->category = flow->guessed_header_category;
      return;
    }
```

categories_loaded flag represents if the initialization of category is finished, as introduced in section 4.1.3. And if the category of this flow is already present, just adopt the old value and return. If not, enter the second part:

Listing 82: 2nd Part of **ndpi_fill_protocol_category**

```
    if(flow->host_server_name[0] != '\0') {
      int rc = ndpi_match_custom_category(ndpi_str, (char *) flow->host_server_name, strlen((
          char *) flow->host_server_name), &id);

      if(rc == 0) {
        flow->category = ret->category = (ndpi_protocol_category_t) id;
        return;
      }
    }
  }
```

The second part uses **host_server_name** to find the category, **host_server_name** is extracted in some dissectors, for example, for HTTP dissector, it will set by the host line in HTTP header.

Then, the ac trie will be search by **ndpi_match_custom_category** function using the **host_server_name**, and if result **rc** is 0, means there is a full match or a partial match in the ac trie, and directly return the searching result.

ndpi_match_custom_category will simply call the **ndpi_match_string_protocol_id** as below:

```
int ndpi_match_string_protocol_id (...)  {
    ...
    rc = ac_automata_search(automa, &ac_input_text, &match);

    if((rc == 0) && (match.number != 0))
        rc = 1;

    if(rc)
        *protocol_id = (u_int16_t)match.number, *category = match.category, *breed = match.breed;
    else
        *protocol_id = NDPI_PROTOCOL_UNKNOWN;

    return((*protocol_id != NDPI_PROTOCOL_UNKNOWN) ? 0 : -1);
}
```

ndpi_match_string_protocol_id search the ac trie through **ac_automata_search** function, which has been introduced in section 3.3.6. **ac_automata_search** will return two results, 0 represent a partial match or no match, and 1 represent a fully match. If 0 is return and also **match.number** is not **NULL**, means a patrial match is detected, then set **rc** to 1 and corresponding protocol result, thus the partial match acts as a fully match in this situation.

Listing 83: 3rd Part of **ndpi_fill_protocol_category**

```
    flow->category = ret->category = ndpi_get_proto_category(ndpi_str, *ret);
}
```

According to the 1st and 2nd part of **ndpi_fill_protocol_category**, when enter the last part, it implies the flow doesn't have history detected category, and also failed to find a node in the ac trie through **host_server_name**. Then **ndpi_get_proto_category** will be used to get the category directly from **proto_defaults**.

```
ndpi_protocol_category_t ndpi_get_proto_category(struct ndpi_detection_module_struct *ndpi_str,
    ndpi_protocol proto) {
    if((proto.master_protocol == NDPI_PROTOCOL_UNKNOWN) ||
            (ndpi_str->proto_defaults[proto.app_protocol].protoCategory !=
                NDPI_PROTOCOL_CATEGORY_UNSPECIFIED)) {
        if(proto.app_protocol < (NDPI_MAX_SUPPORTED_PROTOCOLS +
            NDPI_MAX_NUM_CUSTOM_PROTOCOLS))
            return(ndpi_str->proto_defaults[proto.app_protocol].protoCategory);
    } else if(proto.master_protocol < (NDPI_MAX_SUPPORTED_PROTOCOLS +
        NDPI_MAX_NUM_CUSTOM_PROTOCOLS))
        return(ndpi_str->proto_defaults[proto.master_protocol].protoCategory);

    return(NDPI_PROTOCOL_CATEGORY_UNSPECIFIED);
}
```

It can be seen from this function that the **app_protocol** has higher priority than the **master_protocol**, **app_protocol** will first act as an index to check the **protoCategory** of the corresponding item in **proto_defaults** array. If **protoCategory** is **NDPI_PROTOCOL_CATEGORY_UNSPECIFIED**, **master_protocol** will be try.
The procedure of guessing the category can be summarized in Fig.29.

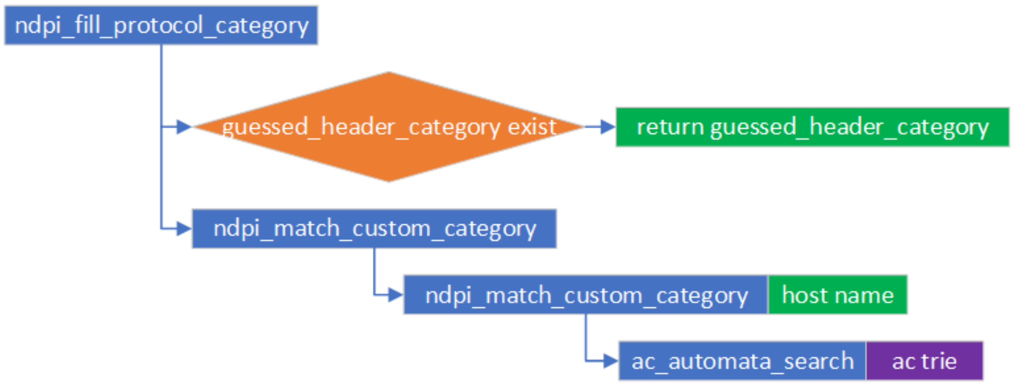

ndpi_fill_protocol_category
guessed_header_category exist
return guessed_header_category
ndpi_match_custom_category
ndpi_match_custom_category
host name
ac_automata_search
ac trie

Figure 29: Procedure of **ndpi_fill_protocol_category**.

5.2.7 Callback Function

Now back to the **ndpi_check_flow_func** function in **ndpi_detection_process_packet**, as below:

```
u_int32_t ndpi_check_flow_func(struct ndpi_detection_module_struct *ndpi_str,
        struct ndpi_flow_struct *flow, NDPI_SELECTION_BITMASK_PROTOCOL_SIZE *
            ndpi_selection_packet) {
    if(!flow)
        return(0);
    else if(flow->packet.tcp != NULL)
        return(check_ndpi_tcp_flow_func(ndpi_str, flow, ndpi_selection_packet));
    else if(flow->packet.udp != NULL)
        return(check_ndpi_udp_flow_func(ndpi_str, flow, ndpi_selection_packet));
    else
        return(check_ndpi_other_flow_func(ndpi_str, flow, ndpi_selection_packet));
}
```

It can be directly seen that there are 3 kinds of function need to be check. One is for TCP, one is for UDP, the others are for neither TCP nor UDP protocol.

The parameter **ndpi_selection_packet** is determined by the packet type calculated in function **ndpi_detection_process_packet**, base on the payload above level 3 layer. Later, **ndpi_selection_packet** will be match to the **ndpi_selection_bitmask** defined in function **ndpi_set_bitmask_protocol_detection** to determine a callback function.

Take HTTP request with payload as an example, when **ndpi_selection_packet** of this packet is detected, **check_ndpi_tcp_flow_func** function can be omit to:

Listing 84: 1st Part of check_ndpi_tcp_flow_func

```
static u_int32_t check_ndpi_tcp_flow_func(struct ndpi_detection_module_struct *ndpi_str,
        struct ndpi_flow_struct *flow,
        NDPI_SELECTION_BITMASK_PROTOCOL_SIZE *ndpi_selection_packet) {

    ...

    u_int16_t proto_index = ndpi_str->proto_defaults[flow->guessed_protocol_id].protoIdx;
    int16_t proto_id = ndpi_str->proto_defaults[flow->guessed_protocol_id].protoId;
    NDPI_SAVE_AS_BITMASK(detection_bitmask, flow->packet.detected_protocol_stack[0]);

    if(flow->packet.payload_packet_len != 0) {
        if((proto_id != NDPI_PROTOCOL_UNKNOWN) && NDPI_BITMASK_COMPARE(flow->
            excluded_protocol_bitmask, ndpi_str->callback_buffer[proto_index].
            excluded_protocol_bitmask) == 0 && NDPI_BITMASK_COMPARE(ndpi_str->
            callback_buffer[proto_index].detection_bitmask, detection_bitmask) != 0 && (ndpi_str->
```

```
                  callback_buffer[proto_index].ndpi_selection_bitmask & *ndpi_selection_packet) == ndpi_str-
                  >callback_buffer[proto_index].ndpi_selection_bitmask) {
              if ((flow->guessed_protocol_id != NDPI_PROTOCOL_UNKNOWN) &&
                      (ndpi_str->proto_defaults[flow->guessed_protocol_id].func != NULL))
                  ndpi_str->proto_defaults[flow->guessed_protocol_id].func(ndpi_str, flow),
                  func = ndpi_str->proto_defaults[flow->guessed_protocol_id].func, num_calls++;
          }

      ...
```

Note the **detected_protocol_stack** indicates the protocol detection result in history, if it is the first time to detect a protocol of the flow, it is **NDPI_PROTOCOL_UNKNOWN**.

It can be known there are 5 conditions to enter the callback function:

1) **guessed_protocol_id** is not **NDPI_PROTOCOL_UNKNOWN**.

2) **excluded_protocol_bitmask** does not be set.

3) **detection_bitmask** always be set.

4) **ndpi_selection_bitmask** match.

5) callback function is not null.

Where the condition 1),5) is obvious.

Condition 2) is unsatisfied when a flow's packet count exceed a pre-defined threshold, as described in **ndpi_search_http_tcp**, when reach the threshold, the callback function will never be call.

the first half of this function use the guessed protocol id to call the corresponding protocol analyzer. It is a optimization, since the **packet.detected_protocol_stack[0]** must be determined first before match the **detection_bitmask**.

Listing 85: 2nd Part of check_ndpi_tcp_flow_func

```
          if (flow->detected_protocol_stack[0] == NDPI_PROTOCOL_UNKNOWN) {
              for (a = 0; a < ndpi_str->callback_buffer_size_tcp_payload; a++) {
                  if ((func != ndpi_str->callback_buffer_tcp_payload[a].func) &&
                          (ndpi_str->callback_buffer_tcp_payload[a].ndpi_selection_bitmask & *
                              ndpi_selection_packet) ==
                          ndpi_str->callback_buffer_tcp_payload[a].ndpi_selection_bitmask &&
                          NDPI_BITMASK_COMPARE(flow->excluded_protocol_bitmask,
                          ndpi_str->callback_buffer_tcp_payload[a].excluded_protocol_bitmask) == 0 &&
                          NDPI_BITMASK_COMPARE(ndpi_str->callback_buffer_tcp_payload[a].
                              detection_bitmask,
                          detection_bitmask) != 0) {
                      ndpi_str->callback_buffer_tcp_payload[a].func(ndpi_str, flow), num_calls++;
                      if (flow->detected_protocol_stack[0] != NDPI_PROTOCOL_UNKNOWN)
                          break; /* Stop after detecting the first protocol */
                  }
              }
          }
      } else {
          ...
      }

      return(num_calls);
  }
```

NDPI_BITMASK_COMPARE return 1 if any position in bitmask **a** and **b** have both set to 1.

If the **packet.detected_protocol_stack[0]** is not determined, or the dissector choose by **guessed_protocol_id** is failed to resolve the packet, the second half of this function traversal all the matched protocol dissectors to do analyzing. All the dissectors which will be used must satisfy 4 conditions:

1) The new selected callback function is not the function in part 1.

2) **excluded_protocol_bitmask** does not be set.

3) **detection_bitmask** always be set.

4) **ndpi_selection_bitmask** match.

The condition 1) is obvious, the condition 2),3),4) is corresponding to the 2),3),4) in the first part. Where the major different between the first part and second part is that the first part using **callback_buffer** to compare the **ndpi_selection_bitmask**, and the second part using the **callback_buffer_tcp_payload**. The key difference about this two structure have been analyzed in previous section.

Suppose an http request with payload and the URL is **"www.baidu.com"**, then it will go into the **func** which is set in **init_http_dissector** with pointer **ndpi_search_http_tcp**, this function will be introduced in section 7.

The procedure of using the callback function in corresponding dissector is in Fig.30.

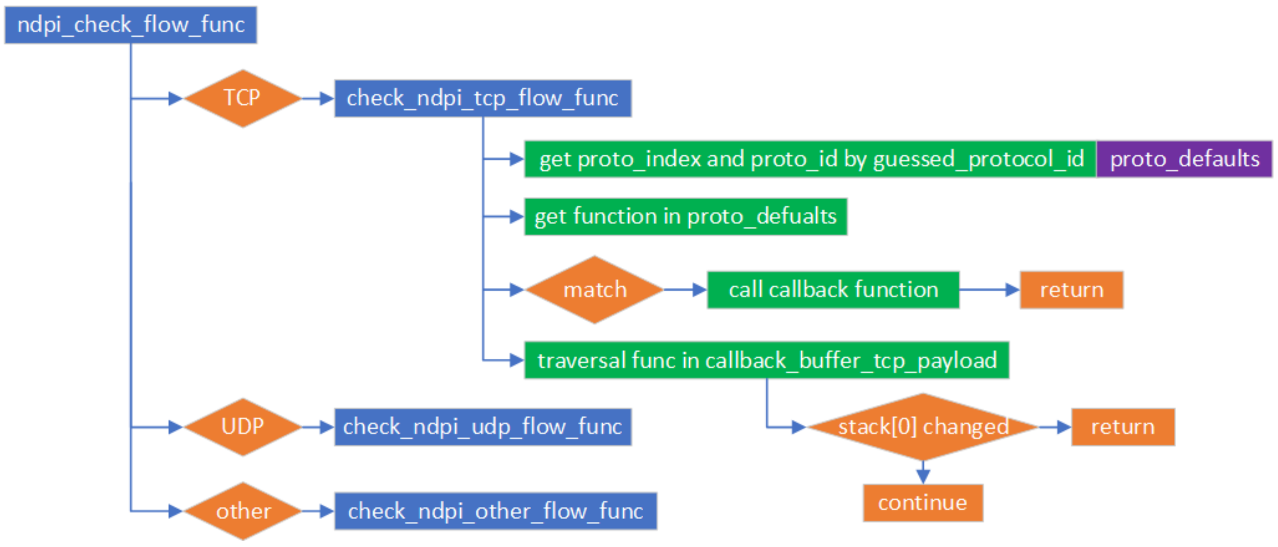


Figure 30: Procedure of **ndpi_check_flow_func**.

5.2.8 Cancel Detection

The detection procedure can be paused by **ndpi_detection_giveup** function for some reasons. The main purpose of this function is to determine the **master_protocol** and **app_protocol** value by existing information before stopping the detection. Furthermore, the history detection result **detected_protocol_stack** will be used if other information collected by current packet is not convinced enough.

Listing 86: 1st Part of **ndpi_detection_giveup**

```
ndpi_protocol ndpi_detection_giveup (...,  u_int8_t  enable_guess,  ...)  {

    ndpi_protocol ret  = {NDPI_PROTOCOL_UNKNOWN, NDPI_PROTOCOL_UNKNOWN,
        NDPI_PROTOCOL_CATEGORY_UNSPECIFIED};

    *protocol_was_guessed = 0;
    ret.master_protocol = flow->detected_protocol_stack[1],  ret.app_protocol = flow->
        detected_protocol_stack[0];
    ret.category  = flow->category;

    if((ret.master_protocol != NDPI_PROTOCOL_UNKNOWN) && (ret.app_protocol !=
        NDPI_PROTOCOL_UNKNOWN))
        return(ret);
```

First take a note to the **enable_guess** parameter. When it is **0**, means the return value will adopt the history detection results, except for some special protocol, when it is **1**, indicates the return value will use the current guessed results, namely the **guessed_host_protocol_id** or **guessed_protocol_id**.

The return result **ndpi_protocol** contains 3 item, the **master_protocol**, the **app_protocol** and the **category**. If the history detection result(**detected_protocol_stack**) contains the **master_protocol** and **app_protocol**, just set the value in **ndpi_protocol** and return.

If the **detected_protocol_stack[0]** is not detected before, it will enter the second part:

Listing 87: 2nd Part of **ndpi_detection_giveup**

```
    if(flow->detected_protocol_stack[0] == NDPI_PROTOCOL_UNKNOWN) {
        ...
        else if(enable_guess){
            ...

            *protocol_was_guessed = 1;
            ndpi_int_change_protocol(ndpi_str, flow, guessed_host_protocol_id, guessed_protocol_id);
        }
    }
```

Ignore some special cases, when **enable_guess** is set, the current guessed result **guessed_protocol_id** and **guessed_host_protocol_id** will replace the history detection result in **ndpi_int_change_protocol**, as below:

Listing 88: 1st Part of **ndpi_int_change_protocol**

```
void ndpi_int_change_protocol (...)  {
    if((upper_detected_protocol == NDPI_PROTOCOL_UNKNOWN) && (lower_detected_protocol !=
        NDPI_PROTOCOL_UNKNOWN))
        upper_detected_protocol = lower_detected_protocol;

    if(upper_detected_protocol == lower_detected_protocol)
        lower_detected_protocol = NDPI_PROTOCOL_UNKNOWN;

    if((upper_detected_protocol != NDPI_PROTOCOL_UNKNOWN) && (lower_detected_protocol ==
        NDPI_PROTOCOL_UNKNOWN)) {
        if((flow->guessed_host_protocol_id != NDPI_PROTOCOL_UNKNOWN) &&
                (upper_detected_protocol != flow->guessed_host_protocol_id)) {
            if(ndpi_str->proto_defaults[upper_detected_protocol].can_have_a_subprotocol) {
                lower_detected_protocol = upper_detected_protocol;
                upper_detected_protocol = flow->guessed_host_protocol_id;
            }
        }
    }
```

The **upper_detected_protocol** corresponding to the **guessed_host_protocol_id**, which is also the **detected_protocol_stack[0]** and guessed by IP address or host name. the **lower_detected_protocol** relates to the **guessed_protocol_id**, which is also the **detected_protocol_stack[1]** and determined by the TCP or UDP port number.

upper_detected_protocol represents more specifically protocol then **lower_detected_protocol**. The general priority criteria is that **upper_detected_protocol** has higher priority than **lower_detected_protocol**. Base on this priority, 3 cases will be handled as below:

- **upper_detected_protocol is NULL and lower_detected_protocol is not null:** in this case, just use the **lower_detected_protocol** result.

- **upper_detected_protocol equals to the lower_detected_protocol:** includes the first case. Base on the priority, **lower_detected_protocol** will be set to **NULL**.
- **upper_detected_protocol is not NULL and lower_detected_protocol is null:** this case includes the second case. It means a possible sub-protocol is detected by the result of **guessed_host_protocol_id**. For examples, **upper_detected_protocol** is **HTTP**, and **guessed_host_protocol_id** is **SKYPE**. Finally, the **upper_detected_protocol** will be set to **SKYPE**, and **lower_detected_protocol** will be the **HTTP**.

Listing 89: 2nd Part of **ndpi_int_change_protocol**

```
        ndpi_int_change_flow_protocol(ndpi_str, flow, upper_detected_protocol, lower_detected_protocol);
        ndpi_int_change_packet_protocol(ndpi_str, flow, upper_detected_protocol, lower_detected_protocol);
    }
```

After determined the **upper_detected_protocol** and **lower_detected_protocol**, the second part of **ndpi_int_change_protocol** function will set the results in history structure. **ndpi_int_change_flow_protocol** set the flow's history structure, as below:

```
    void ndpi_int_change_flow_protocol(struct ndpi_detection_module_struct *ndpi_str, struct
         ndpi_flow_struct *flow, u_int16_t upper_detected_protocol, u_int16_t lower_detected_protocol) {
        flow->detected_protocol_stack[0] = upper_detected_protocol,
        flow->detected_protocol_stack[1] = lower_detected_protocol;
    }
```

While the **ndpi_int_change_packet_protocol** function set the history structure in current packet. as below:

```
    void ndpi_int_change_packet_protocol(struct ndpi_detection_module_struct *ndpi_str, struct
         ndpi_flow_struct *flow, u_int16_t upper_detected_protocol, u_int16_t lower_detected_protocol) {
        struct ndpi_packet_struct *packet = &flow->packet;

        packet->detected_protocol_stack[0] = upper_detected_protocol,
        packet->detected_protocol_stack[1] = lower_detected_protocol;
    }
```

Back to the **ndpi_detection_giveup** function, if **enable_guess** is set, and the flow's history result **detected_protocol_stack[0]** is not NULL, means it is the time to determined if the previous result should be changed.

Listing 90: 3rd Part of **ndpi_detection_giveup**

```
        else if(enable_guess) {
            if(flow->guessed_protocol_id != NDPI_PROTOCOL_UNKNOWN) {
                *protocol_was_guessed = 1;
                flow->detected_protocol_stack[1] = flow->guessed_protocol_id;
            }

            if(flow->guessed_host_protocol_id != NDPI_PROTOCOL_UNKNOWN) {
                *protocol_was_guessed = 1;
                flow->detected_protocol_stack[0] = flow->guessed_host_protocol_id;
            }

            if(flow->detected_protocol_stack[1] == flow->detected_protocol_stack[0]) {
                *protocol_was_guessed = 1;
                flow->detected_protocol_stack[1] = flow->guessed_host_protocol_id;
            }
        }
```

Also 3 cases should be consider:

- **guessed_protocol_id is not NULL:** set to the **detected_protocol_stack**[1].
- **guessed_host_protocol_id is not NULL:** set to the **detected_protocol_stack**[0].
- **detected_protocol_stack**[1] **equals to the detected_protocol_stack**[0]**:** for example, if **detected_protocol_stack**[1] and **detected_protocol_stack**[0] is both **HTTP**, and both are set by the **guessed_host_protocol_id**, in this case, just set **detected_protocol_stack**[1] to **SKYPE**, since **SKYPE** is more specifically.

The last part of **ndpi_detection_giveup** function is as below:

Listing 91: 4th Part of **ndpi_detection_giveup**

```
        ret.master_protocol = flow->detected_protocol_stack[1];
        ret.app_protocol = flow->detected_protocol_stack[0];

        ...

        if(ret.app_protocol != NDPI_PROTOCOL_UNKNOWN) {
            *protocol_was_guessed = 1;
            ndpi_fill_protocol_category(ndpi_str, flow, &ret);
        }

        return(ret);
    }
```

The return **app_protocol** and **master_protocol** is determined and maybe changed in this part. Note that the two **detected_protocol_stack** must exist one or two NULL value. If the **detected_protocol_stack** is **NULL**, the category is calculated in **ndpi_fill_protocol_category**.

ndpi_detection_giveup function can be summarized in Fig.31.

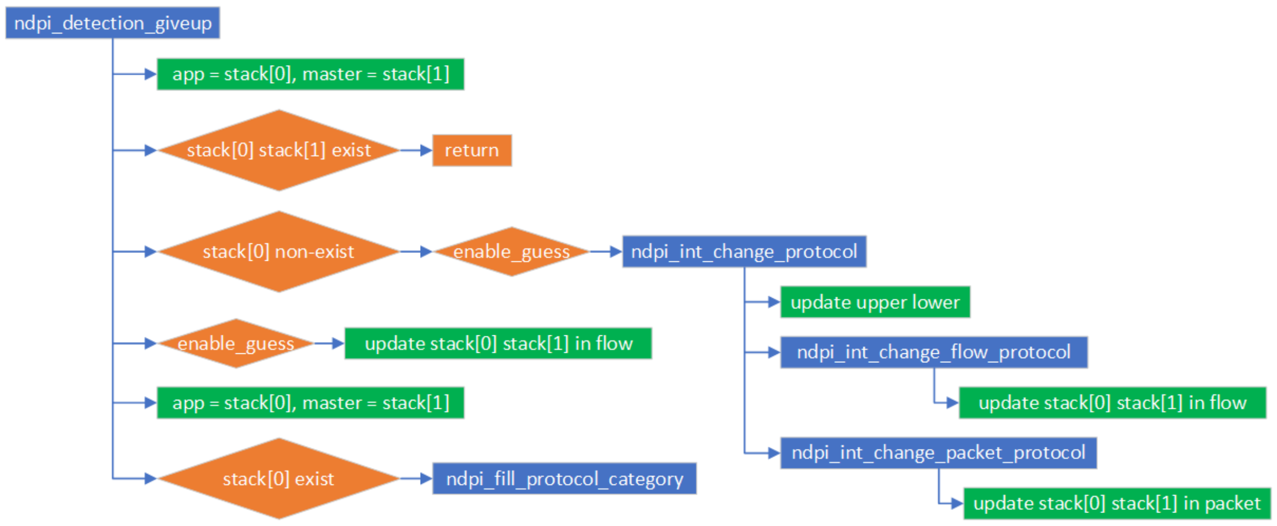

ndpi_detection_giveup
app = stack[0], master = stack[1]
stack[0] stack[1] exist
return
stack[0] non-exist
enable_guess
ndpi_int_change_protocol
update upper lower
ndpi_int_change_flow_protocol
update stack[0] stack[1] in flow
enable_guess
update stack[0] stack[1] in flow
app = stack[0], master = stack[1]
ndpi_int_change_packet_protocol
stack[0] exist
ndpi_fill_protocol_category
update stack[0] stack[1] in packet

Figure 31: Procedure of **ndpi_detection_giveup**.

5.3 Summary

In this section, several key functions when processing a packet is analyzed, includes **ndpi_init_packet_header** function in section 5.2.2, **ndpi_connection_tracking** function in section 5.2.3, **ndpi_detection_process_packet** function in section 5.2.4, **ndpi_guess_host_protocol_id** function in section 5.2.5, **ndpi_fill_protocol_category** function in section 5.2.6, **ndpi_detection_process_packet** function in section 5.2.7, **ndpi_detection_giveup** function in section 5.2.8.

According to section 5.2.1, all this functions can be combined and the procedure of **ndpi_detection_process_packet** function can be concluded in Fig.32.

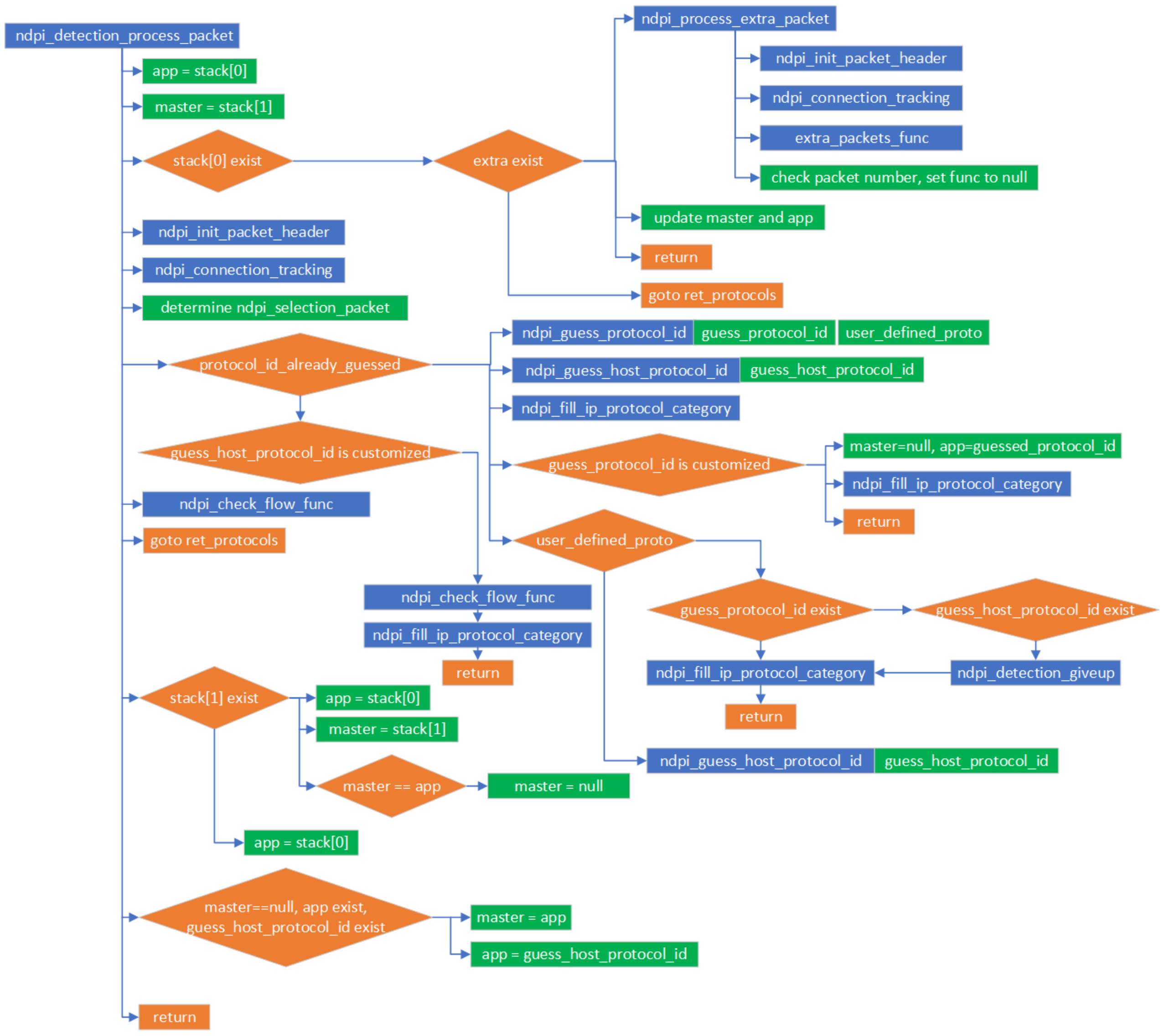


Figure 32: Main procedure of processing packet.

Several branches are explained as:

- **extra branch:** for each flow, if the **detected_protocol_stack[0]**, i.e. the application protocol has been confirmed from previous packets, all later packets will only go into the extra packets handling procedure.
- **guess protocol id:** for each flow, the first packet will be extracted and guessed the protocol ID, if failed, next packet will repeat guessing the protocol ID, if successfully, the **protocol_id_already_guessed** will be set true, and later packets will not guess the protocol ID again.
- **callback function:** as long as the packet doesn't enter the extra branch, each packets will call its corresponding callback function.
- **guess category:** as long as the **protocol_id_already_guessed** flag doesn't be set, category will be guessed.
- **protocol priority:** the application protocol is more specifically than the master protocol, in general, the application protocol relates to **guessed_host_protocol_id**, and it is finally set into the **detect-**

ed_protocol_stack[0], the master protocol relates to the **guessed_protocol_id**, and finally stores into the **detected_protocol_stack[1]**.

6 Subscribe Analyzer

nDPI has support many protocol analyzer, which locate in the **"/src/lib/procotol"**. The procedure of create custom protocol analyzer is as below:

6.1 Define a Protocol ID

The default protocol ID is located in **"/src/include/ndpi_protocol_ids"**, define as **enum ndpi_protocol_id_t**, shown as:

```
typedef enum {
    NDPI_PROTOCOL_UNKNOWN      = 0,
    NDPI_PROTOCOL_FTP_CONTROL = 1,

     ...

    NDPI_PROTOCOL_ANYDESK       = 252,
    NDPI_PROTOCOL_SOAP          = 253,

#ifdef CUSTOM_NDPI_PROTOCOLS
#include " ../../../ nDPI-custom/custom_ndpi_protocol_ids.h"
#endif

    NDPI_LAST_IMPLEMENTED_PROTOCOL
} ndpi_protocol_id_t ;
```

According to above code, user can directly add a new default protocol ID, such as **"NDPI_PROTOCL_INDEX = 254"** among this code, but more standard procedure should be creating **nDPI-custom** directory under nDPI, then create the **customer_ndpi_protocol_ids.h** file under it, and add custom protocol IDs in this file.

As discussed in section 4.3, user can specify a protocol through **"-p"** option and pass the custom protocol file. The new protocol defined in this file is customized protocol, and the protocol ID is above **NDPI_LAST_IMPLEMENTED_PROTOCOL**. As introduced in section 5.2.1, the procedure of the default protocol ID which is smaller than **NDPI_LAST_IMPLEMENTED_PROTOCOL** is different with the customized protocol ID.

6.2 Register Analyzer

ndpi_set_bitmask_protocol_detection function is used to register an dissector, as below:

```
void ndpi_set_bitmask_protocol_detection(struct ndpi_detection_module_struct *ndpi_str, const
    NDPI_PROTOCOL_BITMASK *detection_bitmask, const u_int32_t idx, u_int16_t ndpi_protocol_id,
    void (*func)(struct ndpi_detection_module_struct *, struct ndpi_flow_struct *flow), const
    NDPI_SELECTION_BITMASK_PROTOCOL_SIZE ndpi_selection_bitmask, ...) {

    if (NDPI_COMPARE_PROTOCOL_TO_BITMASK(*detection_bitmask, ndpi_protocol_id) != 0) {

        ndpi_str->proto_defaults[ndpi_protocol_id ]. protoIdx = idx;
        ndpi_str->proto_defaults[ndpi_protocol_id ]. func = ndpi_str->callback_buffer[idx]. func = func;
        ndpi_str->callback_buffer [idx ]. ndpi_protocol_id = ndpi_protocol_id;
        ndpi_str->callback_buffer [idx ]. ndpi_selection_bitmask = ndpi_selection_bitmask;

         ...

        NDPI_ADD_PROTOCOL_TO_BITMASK(ndpi_str->callback_buffer[idx].detection_bitmask,
            ndpi_protocol_id);
        NDPI_SAVE_AS_BITMASK(ndpi_str->callback_buffer[idx].excluded_protocol_bitmask,
            ndpi_protocol_id);
    }
}
```

Where **ndpi_str** is the detection module structure which created by **ndpi_init_detection_module** function; the **detection_bitmask** represents which protocol should be detected, if **ndpi_init_detection_module** is called from ndpiReader, from the definition of **NDPI_BITMASK_SET_ALL** it can be seen all the protocols will be detected; **func** is the callback function which will be finally called if nDPI guesses the corresponding protocol.

First take a look at some micro of this function:

```
#define NDPI_ZERO(p)   memset((char *)(p), 0, sizeof (*(p)))
#define NDPI_SET(p, n)  ((p)->fds_bits[(n)/NDPI_BITS] |= (1ul << (((u_int32_t)n) % NDPI_BITS)))
#define NDPI_ISSET(p, n) ((p)->fds_bits[(n)/NDPI_BITS] & (1ul << (((u_int32_t)n) % NDPI_BITS)))

#define NDPI_COMPARE_PROTOCOL_TO_BITMASK(bmask,value) NDPI_ISSET(&bmask,value)
#define NDPI_SAVE_AS_BITMASK(bmask,value) { NDPI_ZERO(&bmask) ;
    NDPI_ADD_PROTOCOL_TO_BITMASK(bmask, value); }
#define NDPI_ADD_PROTOCOL_TO_BITMASK(bmask,value) NDPI_SET(&bmask,value)
```

It can be seen **NDPI_COMPARE_PROTOCOL_TO_BITMASK** is used to check if the corresponding bit is set in **ndpi_protocol_bitmask_struct**, **NDPI_ADD_PROTOCOL_TO_BITMASK** will set the corresponding bit to 1, different with the micro **NDPI_SAVE_AS_BITMASK**, which clear the **ndpi_protocol_bitmask_struct** before setting the bits.

According to these definition, it can be seen that **ndpi_set_bitmask_protocol_detection** first checks if the protocol(in this environment, it is HTTP) can be add, then, the protocol index, protocol ID, callback function, and selective mask are all added to the **ndpi_str** struct.

Note that the protocol index is not meaningful, it just represents an index in the **callback_buffer** array. When receiving a packet, the protocol ID is guessed an acted as an index to find the protocol index in the **proto_defaults** array, and find the callback function in the **callback_buffer**.

Besides the protocol ID, **ndpi_selection_bitmask** add another condition when checking the callback function. For example, with payload and without payload, with retransmission and without retransmission. The **ndpi_selection_bitmask** is necessary since there are situation where one protocol id relates to many callback function.

Finally, **excluded_protocol_bitmask** is used to skip some protocol detection in some timing.

7 HTTP Dissector

Base on the description of function **func**, after guess the protocol ID, it will go into the callback function, which is set by **ndpi_set_bitmask_protocol_detection**.

According to **init_http_dissector** function, if the HTTP protocol ID is guessed, the corresponding callback function is **ndpi_search_http_tcp**.

In the next subsections, we will deep dive in this function. In general, the 1st sub-section introduces the http dissector threshold; in the 2nd sub-section, the procedure how nDPI handles HTTP protocol will be described.

7.1 HTTP Detection Threshold

The entry of HTTP callback function is defined as below:

```
static void ndpi_search_http_tcp(struct ndpi_detection_module_struct *ndpi_struct, struct
      ndpi_flow_struct *flow){
    if(flow->packet_counter > 20) {
        NDPI_EXCLUDE_PROTO(ndpi_struct, flow);
        http_bitmask_exclude_other(flow);
        return;
    }
    ndpi_check_http_tcp(ndpi_struct, flow);
}
```

packet_counter increases each time when receive a HTTP packet with **payload** in the same flow. It can be seen if the **packet_counter** exceed an threshold(20), **NDPI_EXCLUDE_PROTO** will exclude HTTP protocol then disable callback function for further HTTP packets. The **http_bitmask_exclude_other** relates to the **XBOX** protocol and micro **NDPI_EXCLUDE_PROTO** is shown as:

```
#define NDPI_EXCLUDE_PROTO(mod,flow) ndpi_exclude_protocol(mod, flow,
      NDPI_CURRENT_PROTO, __FILE__, __FUNCTION__, __LINE__)
void ndpi_exclude_protocol (...)  {
    if(protocol_id < NDPI_MAX_SUPPORTED_PROTOCOLS +
        NDPI_MAX_NUM_CUSTOM_PROTOCOLS) {
        NDPI_ADD_PROTOCOL_TO_BITMASK(flow->excluded_protocol_bitmask, protocol_id);
    }
}
```

NDPI_EXCLUDE_PROTO mask the HTTP protocol in **excluded_protocol_bitmask** on this flow, from **check_ndpi_tcp_flow_func** it can be known if **excluded_protocol_bitmask** is mask for some protocol, the callback function will never be used. Then the history result **detected_protocol_stack** would be called instead.

7.2 HTTP Request Analyzer

ndpi_check_http_tcp is the primary function in **ndpi_search_http_tcp**, as below:

Listing 92: 1st part of **ndpi_check_http_tcp**

```
static void ndpi_check_http_tcp(struct ndpi_detection_module_struct *ndpi_struct, struct
      ndpi_flow_struct *flow) {
    ...

    if((packet->payload_packet_len > 0) && (flow->l4.tcp.http_stage == 0)) {
        flow->http_detected = 0;
        filename_start = http_request_url_offset(ndpi_struct, flow);

        if(filename_start == 0) {
            ...
        }

        ndpi_parse_packet_line_info (ndpi_struct, flow);
        ndpi_check_http_header(ndpi_struct, flow);
```

The first part first gets the URL offset inside the http payload, through the **http_request_url_offset** function, which detects the HTTP method(start from the HTTP payload), then use the HTTP method as an index to locate the **http_methods** array, and finally get the offset.

The **http_methods** is as below:

```
http_methods[] = {
            STATIC_STRING_L("GET␣"),
            STATIC_STRING_L("POST␣"),
    ...
```

The offset of each HTTP method is calculated through micro **STATIC_STRING_L**, for example, the offset of **GET** method is 4 and **POST** method is 5.

After determine the payload length, **ndpi_parse_packet_line_info** is used to collect HTTP information from each line in the payload.

ndpi_check_http_header is used to do some security detection.

The **ndpi_parse_packet_line_info** is defined below:

Listing 93: 1st part of **ndpi_parse_packet_line_info**

```
void ndpi_parse_packet_line_info (...) {
    ...

    packet->line[packet->parsed_lines].ptr = packet->payload;

    for (a = 0; ((a+1) < packet->payload_packet_len) && (packet->parsed_lines <
        NDPI_MAX_PARSE_LINES_PER_PACKET); a++) {
        if((packet->payload[a] == 0x0d) && (packet->payload[a+1] == 0x0a)) {
            if(((a + 3) < packet->payload_packet_len)
                && (packet->payload[a+2] == 0x0d)
                && (packet->payload[a+3] == 0x0a)) {
                ...
            }

            ...
```

The first line start from the HTTP payload. The special character "0x0d0a" can split each line. **ndpi_parse_packet_line_info** parses all the lines in HTTP payload, and gets as many line information as it can, take host line as an example:

Listing 94: 2nd part of **ndpi_parse_packet_line_info**

```
            if(packet->line[packet->parsed_lines].len > 6 &&
                strncasecmp((const char *) packet->line[packet->parsed_lines].ptr, "Host:", 5) ==
                    0) {
                packet->host_line.ptr = &packet->line[packet->parsed_lines].ptr[5];
                packet->host_line.len = packet->line[packet->parsed_lines].len - 5;
                packet->http_num_headers++;
            }
            ...
        }
    }
}
```

A host line is recognized by the starting string “Host:”, some special host line with space is omit. After obtain the host line, its start pointer and length are stored in packet.

Many other lines are omit, this includes initial_binary_bytes, response_status_code, server_line, host_line, forwarded_line, content_line, accept_line, referer_line, user_agent_line, http_encoding, http_transfer_encoding, http_contentlen, content_disposition_line, http_cookie, http_origin, http_x_session_type. And other line information such as Date, Vary, ETag, Pragma, Expires, Set|Cookie, Keep|Alive, Connection, Last|Modified, Accept|Ranges, Accept|Language, Accept|Encoding, Upgrade|Insecure|Requests are reserved for future use.

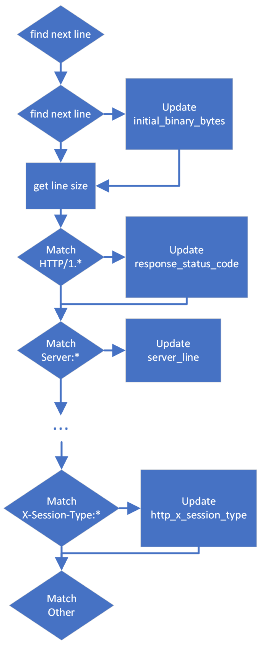


Figure 33: The procedure of ndpi_parse_packet_line_info.

The detail procedure of **ndpi_parse_packet_line_info** is in Fig.33.

Listing 95: 2nd part of **ndpi_check_http_tcp**

```
        if (packet->parsed_lines <= 1) {
            ...
            return;
        }
```

parsed_lines is updated in previous **ndpi_parse_packet_line_info** function. If **“parsed_lines $\leqslant$ 1”**, it means the TCP is in handshake process, thus directly return from this function.

Listing 96: 2nd part of **ndpi_check_http_tcp**

```
        if (packet->line[0].len >= (9 + filename_start)
                && memcmp(&packet->line[0].ptr[packet->line[0].len - 9], " HTTP/1.", 8) == 0) {

            packet->http_url_name.ptr = &packet->payload[filename_start];
            packet->http_url_name.len = packet->line[0].len - (filename_start + 9);

            ...
```

A general http packet will enter this if statement, then some other HTTP information is detected, includes URL, method, version, etc. Part of this code is omit.

Listing 97: 3rd part of **ndpi_check_http_tcp**

```
            ...
            if (packet->host_line.ptr != NULL) {
                ndpi_int_http_add_connection(ndpi_struct, flow, NDPI_PROTOCOL_HTTP,
                    NDPI_PROTOCOL_CATEGORY_WEB);
                check_content_type_and_change_protocol(ndpi_struct, flow);
                return;
            }
        }

        NDPI_EXCLUDE_PROTO(ndpi_struct, flow);
        http_bitmask_exclude_other(flow);
    }
    ...
}
```

Suppose a HTTP packet payload contains the host line, then **ndpi_int_http_add_connection** will be used to set the detection result, then **check_content_type_and_change_protocol** will check the content to see if any change should be make. Moreover, if this is not a standard HTTP packet(HTTP method plus the HTTP/1.1 for example), the detection of this flow will be canceled through **NDPI_EXCLUDE_PROTO** micro.

7.2.1 Initialize HTTP Connection

ndpi_int_http_add_connection will be called only once when a new HTTP connection is detected, shown as:

```
static void ndpi_int_http_add_connection (...) {
    ...
    if ((flow->guessed_host_protocol_id == NDPI_PROTOCOL_UNKNOWN) || (http_protocol !=
        NDPI_PROTOCOL_HTTP))
        flow->guessed_host_protocol_id = http_protocol;

    ndpi_set_detected_protocol(ndpi_struct, flow, flow->guessed_host_protocol_id,
        NDPI_PROTOCOL_HTTP);

    flow->check_extra_packets = 1;
    flow->max_extra_packets_to_check = 5;
    flow->extra_packets_func = ndpi_search_http_tcp_again;
    flow->http_detected = 1;
}
```

For a new HTTP connection, if it is failed to detect **guessed_host_protocol_id** by IP address, meanwhile if the parameter **http_protocol** is a application protocol, **guessed_host_protocol_id** will be set to it.

Then **ndpi_set_detected_protocol** will be used to updated the **detected_protocol_stack** in both flow and packet.

Finally, some variable relates to the extra packets detection will be set, which is introduced in section 5.2.1.

ndpi_set_detected_protocol is as below:

```
void ndpi_set_detected_protocol(struct ndpi_detection_module_struct *ndpi_str, struct ndpi_flow_struct
      *flow, u_int16_t upper_detected_protocol, u_int16_t lower_detected_protocol) {
    ndpi_int_change_protocol(ndpi_str, flow, upper_detected_protocol, lower_detected_protocol);
}
```

ndpi_int_change_protocol has been analyzed in section 5.2.8. As a result, the **detected_protocol_stack**[0] corresponding to the application protocol will be **guessed_host_protocol_id** or **NDPI_PROTOCOL_HTTP**, depends on if **guessed_host_protocol_id** exists; and the **detected_protocol_stack**[1] corresponding to the master protocol will be **NDPI_PROTOCOL_HTTP**.

7.2.2 Guess Protocol By Content

check_content_type_and_change_protocol will resolve the content in HTTP, try to detect the application protocol.

Listing 98: 1st part of **check_content_type_and_change_protocol**

```
static void check_content_type_and_change_protocol(...) {
    ...
    ndpi_set_detected_protocol(ndpi_struct, flow, NDPI_PROTOCOL_HTTP,
        NDPI_PROTOCOL_UNKNOWN);
```

First set the default detection result through **ndpi_set_detected_protocol** function, which has been analyzed in section 7.2.1. It can be seen if nothing is detected through the HTTP content, this result is reserved and **NDPI_PROTOCOL_HTTP** will be the application protocol.

Listing 99: 2nd part of **check_content_type_and_change_protocol**

```
    if(packet->server_line.ptr != NULL && (packet->server_line.len > 7)) {
        if(strncmp((const char *)packet->server_line.ptr, "ntopng ", 7) == 0)
            ndpi_set_detected_protocol(ndpi_struct, flow, NDPI_PROTOCOL_NTOP,
                NDPI_PROTOCOL_HTTP);
    }

    ...
```

The second part is a corner case, which check if the **server_line** contains **"ntopng"**, if it is, set the master protocol to **NDPI_PROTOCOL_HTTP** and application protocol to **NDPI_PROTOCOL_NTOP**.

Some part is omit, this includes parsing the full URL, http agent.

Listing 100: 3rd part of **check_content_type_and_change_protocol**

```
    if(packet->host_line.ptr != NULL) {
        strncpy((char*)flow->host_server_name, (char*)packet->host_line.ptr, len);
        ndpi_check_dga_name(ndpi_struct, flow, (char*)flow->host_server_name, 1);
        ndpi_http_parse_subprotocol(ndpi_struct, flow);
```

This part first copy the **host_line** to **host_server_name**, then using**ndpi_check_dga_name** function for attack detection, finally call the **ndpi_http_parse_subprotocol** function, which simply uses the **ndpi_match_host_subprotocol** function, as below:

```
u_int16_t ndpi_match_host_subprotocol(...) {
    ...
    ndpi_automa_match_string_subprotocol(ndpi_str, flow, string_to_match, string_to_match_len,
        master_protocol_id, ret_match, 1);
    ndpi_get_custom_category_match(ndpi_str, string_to_match, string_to_match_len, &id);
    ...
}
```

Two main tasks are in this function, the first task is to find the application protocol ID, the second one is to update the category, both by matching the host name(**string_to_match**). We deep dive into the first searching process, and the other is similar.

ndpi_automa_match_string_subprotocol can be divided into 3 part, as below:

Listing 101: 1st part of **ndpi_automa_match_string_subprotocol**

```
static u_int16_t ndpi_automa_match_string_subprotocol(...) {

    matching_protocol_id = ndpi_match_string_subprotocol(ndpi_str, string_to_match,
        string_to_match_len, ret_match, is_host_match);
```

ndpi_match_string_subprotocol take the **string_to_match** as an input to search the ac trie.

Note that **is_host_match** is used to select which ac trie to match. The **host_automa** which is created base on the URL and the **content_automa** which is constructed by the key content, while the latter is for further usage.

Listing 102: 2nd part of **ndpi_automa_match_string_subprotocol**

```
    if(matching_protocol_id != NDPI_PROTOCOL_UNKNOWN) {
      packet->detected_protocol_stack[1] = master_protocol_id,
      packet->detected_protocol_stack[0] = matching_protocol_id;

      flow->detected_protocol_stack[0] = packet->detected_protocol_stack[0],
      flow->detected_protocol_stack[1] = packet->detected_protocol_stack[1];

      if(flow->category == NDPI_PROTOCOL_CATEGORY_UNSPECIFIED)
        flow->category = ret_match->protocol_category;

      return(packet->detected_protocol_stack[0]);
    }
```

If successfully find the **matching_protocol_id**, it must be a new application protocol. Then update the packet's and flow's **detected_protocol_stack** structure.

If the category is not detected, it can be set by the **protocol_category** of matched node. Finally return the application protocol.

Listing 103: 3rd part of **ndpi_automa_match_string_subprotocol**

```
    ret_match->protocol_id = NDPI_PROTOCOL_UNKNOWN, ret_match->protocol_category =
      NDPI_PROTOCOL_CATEGORY_UNSPECIFIED,
    ret_match->protocol_breed = NDPI_PROTOCOL_UNRATED;

    return(NDPI_PROTOCOL_UNKNOWN);
  }
```

When entering the last part, means it is failed to match a node in the ac trie, thus set the **ret_match** and return.

Listing 104: 4th part of **check_content_type_and_change_protocol**

```
      ...

      if((flow->detected_protocol_stack[0] == NDPI_PROTOCOL_UNKNOWN) && (flow->
        http_detected)
          && (packet->http_origin.len > 0)) {
        ndpi_match_host_subprotocol(ndpi_struct, flow, (char *)packet->http_origin.ptr, packet->
          http_origin.len, &ret_match, NDPI_PROTOCOL_HTTP);
      }
```

Back to the **check_content_type_and_change_protocol** function, if fail to update the application protocol by matching the host line in above part of code. nDPI will try using the origin URL, which is contained under the **"Original"** tag, to find the application protocol.

ndpi_match_host_subprotocol function has been analyzed in previous part.

Listing 105: 5th part of **check_content_type_and_change_protocol**

```
      if(flow->detected_protocol_stack[0] != NDPI_PROTOCOL_UNKNOWN) {
        if(packet->detected_protocol_stack[0] != NDPI_PROTOCOL_HTTP) {
          ndpi_int_http_add_connection(ndpi_struct, flow, packet->detected_protocol_stack[0],
            NDPI_PROTOCOL_CATEGORY_WEB);
          return;
```

```
                }
            }
        }
```

This part of code check if successfully finding a application protocol, if so, using **ndpi_int_http_add_connection** to copy the packet's **detected_protocol_stack[0]** to the flow's **detected_protocol_stack[0]**, which represents the application protocol.

Listing 106: 6th part of **check_content_type_and_change_protocol**

```
        if(packet->content_line.ptr != NULL && packet->content_line.len != 0) {
            ndpi_match_content_subprotocol(ndpi_struct, flow,
                                          (char*)packet->content_line.ptr, packet->content_line.len,
                                          &ret_match, NDPI_PROTOCOL_HTTP);

        }

        ndpi_int_http_add_connection(ndpi_struct, flow, packet->detected_protocol_stack[0],
            NDPI_PROTOCOL_CATEGORY_WEB);
    }
```

If entering the final part, means all the method in previous part failed to find the application protocol, the last part check if the content line exists, if so, calling the **ndpi_match_content_subprotocol** for last try. And finally update the application protocol through **ndpi_int_http_add_connection** function, as introduced in section 7.2.1.

ndpi_match_content_subprotocol function will simply call the **ndpi_automa_match_string_subprotocol** function, which match the host name ac trie, not the content ac trie.

7.3 Summary

The procedure of HTTP callback function **ndpi_search_http_tcp** is as Fir.34:

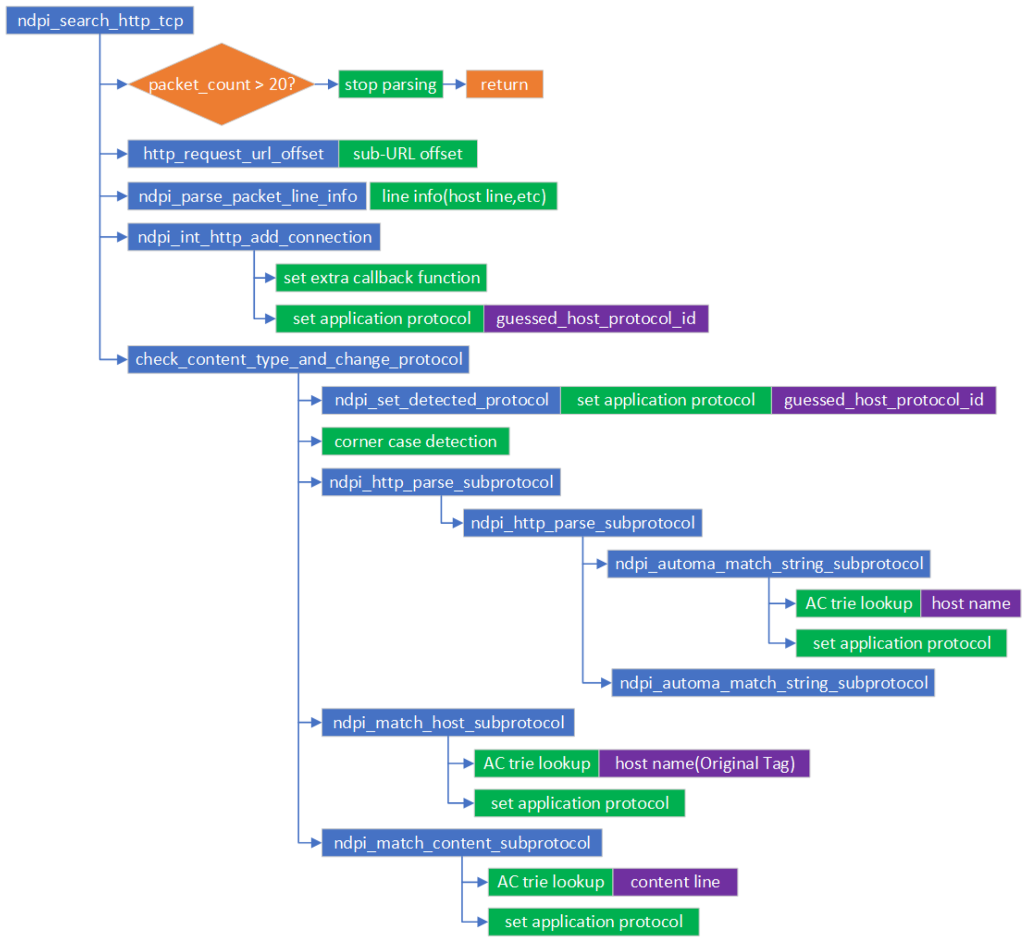


Figure 34: The procedure HTTP dissector.

Some corner case is omit. It can be seen **ndpi_search_http_tcp** will try 4 times to get the application protocol for HTTP packet. The default value is set by **guessed_host_protocol_id**, which is found through patricia tree base on source or destination IP address. The second trial will search the host name ac trie according to the host name. If failed, the third trial will check if the HTTP packet contains the **Original** tag, then replace the host name and try again. Finally, if 2nd and 3rd trial are all failed, nDPI will match the content line to the ac trie to the host name ac trie, and it is the lowest priority.

8 TLS Dissector

This section introduces the TLS dissector. Section 8.1 introduces the initialization of this dissector; section 8.2 analyzes the callback function; a summarization is given in section 8.3.

8.1 Initialization

As introduced in section 4.2, each protocol relates to one or more dissectors, which are initialized in the function **ndpi_set_protocol_detection_bitmask2**.

For TLS protocol, the initialization function is **init_tls_dissector**, as below:

```
void init_tls_dissector (struct ndpi_detection_module_struct *ndpi_struct, u_int32_t *id,
    NDPI_PROTOCOL_BITMASK *detection_bitmask) {
    ndpi_set_bitmask_protocol_detection("TLS", ndpi_struct, detection_bitmask, *id,
        NDPI_PROTOCOL_TLS, ndpi_search_tls_wrapper,
        NDPI_SELECTION_BITMASK_PROTOCOL_V4_V6_TCP_WITH_PAYLOAD_WITHOUT_RETRANSMISSIO
        , SAVE_DETECTION_BITMASK_AS_UNKNOWN, ADD_TO_DETECTION_BITMASK);

    *id += 1;

    ndpi_set_bitmask_protocol_detection("TLS", ndpi_struct, detection_bitmask, *id,
        NDPI_PROTOCOL_TLS, ndpi_search_tls_wrapper,
        NDPI_SELECTION_BITMASK_PROTOCOL_V4_V6_UDP_WITH_PAYLOAD,
        SAVE_DETECTION_BITMASK_AS_UNKNOWN, ADD_TO_DETECTION_BITMASK);

    *id += 1;
}
```

Two callback function is registered for TLS, one is for TCP and the other is for UDP. The callback functions for these two level 4 protocols are specified inside the **ndpi_search_tls_wrapper** function. And the other difference is **ndpi_selection_bitmask** in **callback_buffer**, which is used to specify a kind of callback buffer. For simply, TLS based on TCP protocol is only introduced.

```
static void ndpi_search_tls_wrapper(struct ndpi_detection_module_struct *ndpi_struct,
                                    struct ndpi_flow_struct *flow) {
    ...

    if(packet->udp != NULL)
        ndpi_search_tls_udp(ndpi_struct, flow);
    else
        ndpi_search_tls_tcp(ndpi_struct, flow);
}
```

For TCP packet, **ndpi_search_tls_tcp** will be called, which will be introduced in next section.

8.2 Processing

For the sake of simplicity, only the code relates to protocol detection will be shown, the detail of TLS protocol is in [3].

Listing 107: 1st part of **ndpi_search_tls_tcp**

```
static int ndpi_search_tls_tcp (...) {
    ...
    while(!something_went_wrong) {
        ...
        content_type = flow->l4.tcp.tls.message.buffer[0];

```

```
            if ((len  > 9) && (content_type != 0x17) && (!flow->l4.tcp.tls. certificate_processed )) {
                ...
                processTLSBlock(ndpi_struct, flow);
            } else {
                ...
            }

            ...
        }
        return(1);
    }
```

The start position of TLS buffer is the **content_type**, a **handshake** content type(22) will finally call the **processTLSBlock** function as below:

```
    static int processTLSBlock(...) {
        ...
        switch(packet->payload[0]) {
            case 0x01:
            case 0x02:
                ...
                ndpi_int_tls_add_connection (ndpi_struct,  flow,  NDPI_PROTOCOL_TLS);
                break;
            ...
        }

        return(0);
    }
```

In this function, **payload**[**0**] is the start position of **handshark** payload. **0x01** represents the client hello packet, and **0x02** indicates the server hello packet.

In both cases, the **ndpi_int_tls_add_connection** function is used to set the detection result **detected_protocol_stack** as below:

```
    static void ndpi_int_tls_add_connection (struct ndpi_detection_module_struct *ndpi_struct,
                                              struct ndpi_flow_struct *flow, u_int32_t protocol) {
        ...

        protocol = ndpi_tls_refine_master_protocol (ndpi_struct,  flow,  protocol);
        ndpi_set_detected_protocol (ndpi_struct,  flow,  protocol,  NDPI_PROTOCOL_TLS);
        tlsInitExtraPacketProcessing(ndpi_struct,  flow);
    }
```

ndpi_tls_refine_master_protocol first check if the application protocol should be changed(though the function name is **master protocol**) from **NDPI_PROTOCOL_TLS** according to some special port number. For example, for TLS on TCP port 465, the application protocol will be **NDPI_PROTOCOL_MAIL_SMTPS**, and the master protocol remains **NDPI_PROTOCOL_TLS**.

ndpi_set_detected_protocol function changes the **detected_protocol_stack** as resolved in previous section 7.2.1.

Finally, **tlsInitExtraPacketProcessing** is used to set the extra packets detection.

```
    static void tlsInitExtraPacketProcessing (...)  {
        flow->check_extra_packets = 1;
        flow->max_extra_packets_to_check = 12 + (ndpi_struct->num_tls_blocks_to_follow*4);
        flow->extra_packets_func = (flow->packet.udp != NULL) ? ndpi_search_tls_udp : ndpi_search_tls_tcp
            ;
    }
```

12 maximum extras packets will be resolved through either **ndpi_search_tls_udp** or **ndpi_search_tls_tcp** function. The extra function and related variables of each dissector are visited as described in the beginning of section 5.2.1.

8.3 Summary

More similar than HTTP dissector as introduced in section 7. TLS dissecotr search the client hello or server hello packets, if found, just setting the master protocol to **NDPI_PROTOCOL_TLS**, and if the TCP port is not specified, its application protocol is also **NDPI_PROTOCOL_TLS**.

9 HyperScan

Hyperscan is a high-performance multiple regex matching library available as open source with a C API. Hyperscan uses hybrid automata techniques to allow simultaneous matching of large numbers of regular expressions across streams of data.

nDPI 3.2 support integrated with hyperscan, but nDPI 3.4 remove it to simplify the code. For analysis, this section back to the 3.2 version to see how nDPI use hyperscan.

9.1 Initialization

In nDPI 3.2 version, **init_hyperscan** is the initialization function for hyperscan. If hyperscan is enable, **init_hyperscan** is called from **init_string_based_protocols** function, which is discussed in section 4.1.1.

Listing 108: 1st part of **init_hyperscan**

```
static int init_hyperscan(struct ndpi_detection_module_struct *ndpi_mod) {
    ndpi_mod->hyperscan = (void*)malloc(sizeof(struct hs));
    for (i = 0, j = 0; host_match[i].string_to_match != NULL || host_match[i].pattern_to_match !=
        NULL; i++) {
        if (host_match[i].pattern_to_match) {
            expressions[j] = host_match[i].pattern_to_match;
            ids[j] = host_match[i].protocol_id;
            ++j;
        } else {
            ...
        }
    }
    rc = hyperscan_load_patterns(hs, j, (const char**)expressions, ids);
    return(rc);
}
```

The first part of **init_hyperscan** function adds each protocol ID to the **ids** array, also adds **string_to_match** or **pattern_to_match** to the **expressions**, then call **hyperscan_load_patterns** function to compile this information.

```
static int hyperscan_load_patterns(...) {
    hs_compile_multi(expressions, NULL, ids, num_patterns, HS_MODE_BLOCK, NULL,
                &hs->database, &compile_err);


    hs->scratch = NULL;
     hs_alloc_scratch(hs->database, &hs->scratch);

    return 0;
}
```

hs_compile_multi function compiles the information in **expressions** and **ids** into the hyperscan database. **hs_alloc_scratch** function allocates some temp memory for further calculation.

9.2 Processing

In some of dissectors, **ndpi_automa_match_string_subprotocol** will be called. For example, as discussion in section 7, in HTTP dissector, **ndpi_automa_match_string_subprotocol** maybe be called to extract some information from the **Original** tag, as below:

```
static int ndpi_automa_match_string_subprotocol(...) {
    ...

```

```
        status = hs_scan(hs->database, string_to_match, string_to_match_len, 0, hs->scratch,
                    hyperscanEventHandler, &matching_protocol_id);

        ret_match->protocol_id = matching_protocol_id,
        ret_match->protocol_category = ndpi_struct->proto_defaults[matching_protocol_id].protoCategory,
        ret_match->protocol_breed = ndpi_struct->proto_defaults[matching_protocol_id].protoBreed;

        if (matching_protocol_id != NDPI_PROTOCOL_UNKNOWN) {
            ...
        }
    }
```

When hyperscan is enable, **ndpi_automa_match_string_subprotocol** will call **hs_scan** function to find a match result, in nDPI, the result is the protocol ID, which is stored by **hs_compile_multi** function. The callback function **hyperscanEventHandler** simply stores the result of protocol ID in the variable **matching_protocol_id**. If it exists, the omit part saves this protocol ID to the **detected_protocol_stack**.

10 Memory Management

In section 4.1, **set_ndpi_malloc** and **set_ndpi_flow_malloc** functions are introduced.

In **set_ndpi_malloc** function, **ndpi_malloc_wrapper** and **free_wrapper** wrapper the default **malloc** and **free** function in glibc. While **free_wrapper** simply call the **free** function.

10.1 Wrapper malloc

ndpi_malloc_wrapper is as below:

```
static void *ndpi_malloc_wrapper(size_t size) {
    current_ndpi_memory += size;

    if(current_ndpi_memory > max_ndpi_memory)
        max_ndpi_memory = current_ndpi_memory;

    return(malloc(size));
}
```

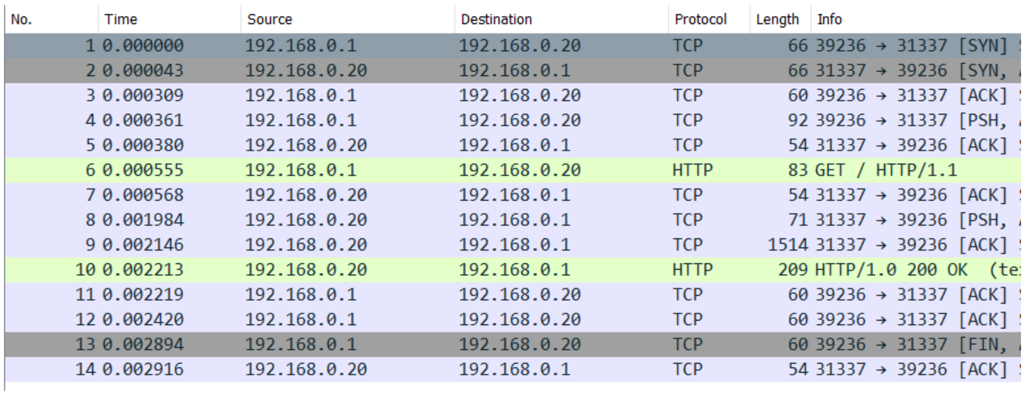
It can be seen ndpi simply wrapper the **malloc** function with two variables: **current_ndpi_memory** and **max_ndpi_memory**. They represent the actual memory usage and peak memory usage when running the program, and just for print usage, which introduced in section 2.3.3.

11 Example

In this section, we will provide some examples about the analysis in nDPI by using ndpiReader. Some example pcap files, which are located in **"tests/pcap"** will be analyzed.

11.1 HTTP example

The processing of pcap file **"tests/pcap/http_lines_split.pcap"** is described in this section. Open it in wireshark gives the view in Fig.35.



No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	192.168.0.1	192.168.0.20	TCP	66	39236 → 31337 [SYN]
2	0.000043	192.168.0.20	192.168.0.1	TCP	66	31337 → 39236 [SYN,
3	0.000309	192.168.0.1	192.168.0.20	TCP	60	39236 → 31337 [ACK]
4	0.000361	192.168.0.1	192.168.0.20	TCP	92	39236 → 31337 [PSH,
5	0.000380	192.168.0.20	192.168.0.1	TCP	54	31337 → 39236 [ACK]
6	0.000555	192.168.0.1	192.168.0.20	HTTP	83	GET / HTTP/1.1
7	0.000568	192.168.0.20	192.168.0.1	TCP	54	31337 → 39236 [ACK]
8	0.001984	192.168.0.20	192.168.0.1	TCP	71	31337 → 39236 [PSH,
9	0.002146	192.168.0.20	192.168.0.1	TCP	1514	31337 → 39236 [ACK]
10	0.002213	192.168.0.20	192.168.0.1	HTTP	209	HTTP/1.0 200 OK (te
11	0.002219	192.168.0.1	192.168.0.20	TCP	60	39236 → 31337 [ACK]
12	0.002420	192.168.0.1	192.168.0.20	TCP	60	39236 → 31337 [ACK]
13	0.002894	192.168.0.1	192.168.0.20	TCP	60	39236 → 31337 [FIN,
14	0.002916	192.168.0.20	192.168.0.1	TCP	54	31337 → 39236 [ACK]

Figure 35: All packets of **"tests/pcap/http_lines_split.pcap"** in wireshark.

Total 14 packets are present in the wireshark. The first 3 packets are the TCP handshake packets. The 4th packet and 6th packet are 2 TCP fragments, which combine to a HTTP request. The 8th to 10th packets are 3 TCP fragments, generate a HTTP response. The processing procedure of this pcap file is as below:

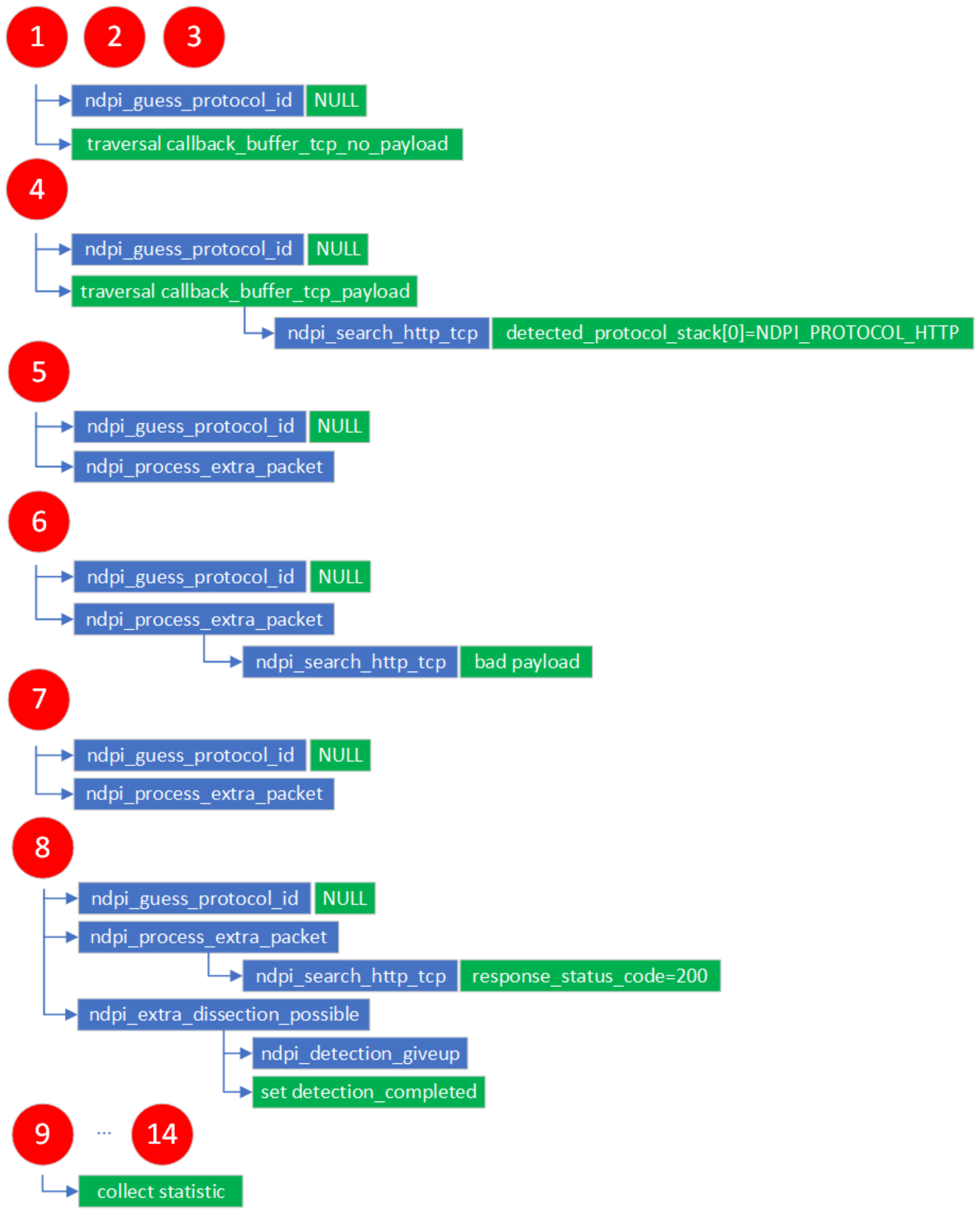


Figure 36: Processing flow of **"tests/pcap/http_lines_split.pcap"**, the red circle represents the packet index.

It can be seen for all the 14 packets, since the port number is not the default 80 for HTTP protocol, and the IP address is picked unrelated to any protocol, the **ndpi_guess_protocol_id** function always fail to guess the protocol(This also includes the **ndpi_guess_host_protocol_id** function, for simply, only one of them is draw).

After failing to guess the protocol, since the 1-3th packets don't contain the payload, they will traversal the **callback_buffer_tcp_no_payload** to find a appropriate callback function, and these callback functions will return nothing.

For the 4th packet, since it is the first fragment of HTTP request, the payload is contained in this packet. Then **callback_buffer_tcp_payload** will be traversal, then **ndpi_search_http_tcp** for HTTP protocol will be used, also, the first fragment contains the HTTP method and HOST information, finally **NDPI_PROTOCOL_HTTP** will be determined and store in the **detected_protocol_stack**.

Since the application protocol is successfully guessed in the 4th packet. For the 5th and 6th packets,

ndpi_process_extra_packet will be called, since the 5th packet doesn't contain the payload, nothing will happen. Also for 6th packet which contains the second fragment of the HTTP request, the payload will be resolved in **ndpi_search_http_tcp** function. But since the second fragment doesn't start with the HTTP method information, this callback function will treat the payload as non-HTTP payload, then directlly return.

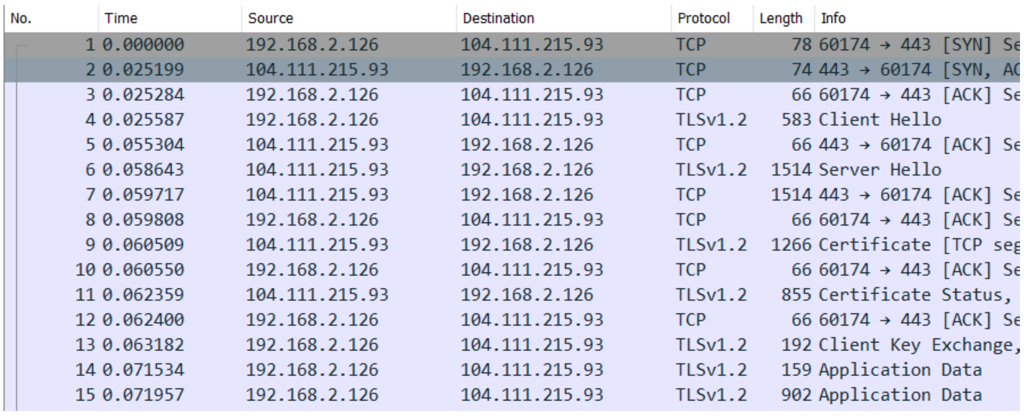
The 7th packets is the same as 5th packets since it doesn't contain the payload.

After that, the first fragment of HTTP response in the 8th packet is handle in the **ndpi_process_extra_packet** function, and finally call the **ndpi_search_http_tcp** function, which detects the response status code 200. S-ince the response status code is found, **ndpi_extra_dissection_possible** will be called, finally the detection is cancel through **ndpi_detection_giveup** and the flag **detection_completed** is set.

Since the **detection_completed** is set after 8th packet. All the remaining packets will do nothing and only collect some statistic for printing.

11.2 TLS example

The example pcap file **"tests/pcap/tls_long_cert.pcap"** will be analyzed in this section. Open it in wire-shark gives the view in Fig.37.



No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	192.168.2.126	104.111.215.93	TCP	78	60174 → 443 [SYN] Se
2	0.025199	104.111.215.93	192.168.2.126	TCP	74	443 → 60174 [SYN, AC
3	0.025284	192.168.2.126	104.111.215.93	TCP	66	60174 → 443 [ACK] Se
4	0.025587	192.168.2.126	104.111.215.93	TLSv1.2	583	Client Hello
5	0.055304	104.111.215.93	192.168.2.126	TCP	66	443 → 60174 [ACK] Se
6	0.058643	104.111.215.93	192.168.2.126	TLSv1.2	1514	Server Hello
7	0.059717	104.111.215.93	192.168.2.126	TCP	1514	443 → 60174 [ACK] Se
8	0.059808	192.168.2.126	104.111.215.93	TCP	66	60174 → 443 [ACK] Se
9	0.060509	104.111.215.93	192.168.2.126	TLSv1.2	1266	Certificate [TCP seg
10	0.060550	192.168.2.126	104.111.215.93	TCP	66	60174 → 443 [ACK] Se
11	0.062359	104.111.215.93	192.168.2.126	TLSv1.2	855	Certificate Status,
12	0.062400	192.168.2.126	104.111.215.93	TCP	66	60174 → 443 [ACK] Se
13	0.063182	192.168.2.126	104.111.215.93	TLSv1.2	192	Client Key Exchange,
14	0.071534	192.168.2.126	104.111.215.93	TLSv1.2	159	Application Data
15	0.071957	192.168.2.126	104.111.215.93	TLSv1.2	902	Application Data

Figure 37: Part of **"tests/pcap/tls_long_cert.pcap"** view in wireshark.

There are totally 182 packets in **"tests/pcap/tls_long_cert.pcap"** file. No.0 to No.2 packets responding to TCP connection establishment. No.4 is the client hello packet in TLS protocol, and No.5 is the corresponding ACK packet. Then No.6 is the server hello packet. After the certificate exchange procedure in No.9, No.11 and No.13 packets. The application data transfer in No.14 and No.15 packets. Later packets is omit.

The procedure of detect the TLS protocol in this pcap file is summarized in Fig.38.

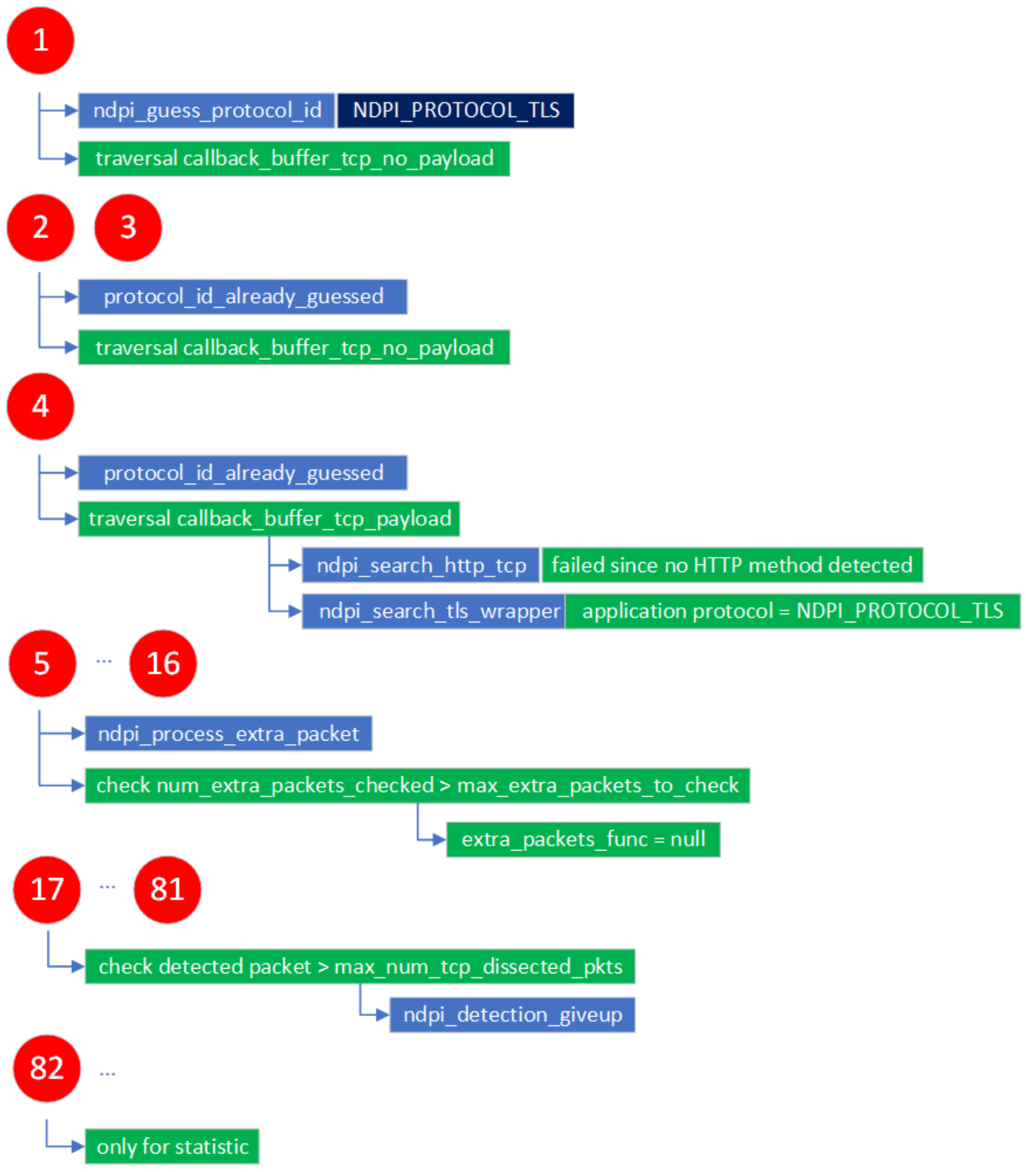

1
ndpi_guess_protocol_id
NDPI_PROTOCOL_TLS
traversal callback_buffer_tcp_no_payload
2
3
protocol_id_already_guessed
traversal callback_buffer_tcp_no_payload
4
protocol_id_already_guessed
traversal callback_buffer_tcp_payload
ndpi_search_http_tcp
failed since no HTTP method detected
ndpi_search_tls_wrapper
application protocol = NDPI_PROTOCOL_TLS
5
...
16
ndpi_process_extra_packet
check num_extra_packets_checked > max_extra_packets_to_check
extra_packets_func = null
17
...
81
check detected packet > max_num_tcp_dissected_pkts
ndpi_detection_giveup
82
...
only for statistic

Figure 38: Processing flow of **"tests/pcap/tls_long_cert.pcap"**, the red circle represents the packet index.

Beginning of the 1st packet, suppose it is the first packet of a new flow, then a flow structure is created. Then the protocol ID **NDPI_PROTOCOL_TLS** is guessed through port number 443. Note that the guessed protocol yet record in the **detected_protocol_stack**. Continue process, the callback function will be search and used to determined the final protocol in **ndpi_check_flow_func**, but according to **init_tls_dissector** function, the **ndpi_selection_bitmask** in either TCP or UDP packet must contains the payload, while the 1st packet doesn't have payload, thus the corresponding callback function would not be used. According to **check_ndpi_tcp_flow_func** function, **callback_buffer_tcp_no_payload** will be traversal, but since the TLS callback function or HTTP callback function is not in this array. Nothing will happen.

After that, the processing of 2nd and 3rd packets will be simple, since the protocol ID is already guessed in the 1st packet, the flag **protocol_id_already_guessed** is set to 1 as described in the **ndpi_process_extra_packet** function, thus it will not be guessed again. The later processing is similar to 1st packet.

For the 4th packet, the first half processing is the same as 2nd and 3rd packets, but in **ndpi_check_flow_func** function, the callback function **ndpi_search_http_tcp** for HTTP and **ndpi_search_tls_wrapper** for TLS will be called. Note that, the default callback function in , Since this is a **Client Hello** packet, as analyzed in section 8.2, the **detected_protocol_stack** is determined in this function. Finally, the application protocol is **NDPI_PROTOCOL_TLS** and the master protocol is **NULL**.

Since the protocol is determined in 4th packet, the 5th to 16th packets will go into the extra packets procedure, as described in section 5.2.1. For the 16th packets, the **num_extra_packets_checked** will be larger than **max_extra_packets_to_check** which is checked in **ndpi_process_extra_packet** function, then **extra_packets_func** will be set to **NULL**.

As described in above paragraph, for 17th to 81th packet, the extra function will not be called again. Finally, the detected packets number will be larger than **max_num_tcp_dissected_pkts**, which is denoted in **packet_processing** function of ndpiReader. The only remaining job is to collect the statistic for each packet in this flow.

12 Problem Explanation

12.1 Question 1: What happen if different protocols with the same port number on the binary sort tree.

This question comes from the code below:

```
static void ndpi_init_protocol_defaults (struct ndpi_detection_module_struct *ndpi_str) {
    ...
    ndpi_set_proto_defaults (ndpi_str, NDPI_PROTOCOL_SAFE, NDPI_PROTOCOL_FBZERO, 0,
        no_master, no_master, "FacebookZero",
        NDPI_PROTOCOL_CATEGORY_SOCIAL_NETWORK, ndpi_build_default_ports(ports_a, 443,
        0, 0, 0, 0), ndpi_build_default_ports(ports_b, 0, 0, 0, 0, 0));
    ndpi_set_proto_defaults (ndpi_str, NDPI_PROTOCOL_SAFE, NDPI_PROTOCOL_TLS, 1,
        no_master, no_master, "TLS", NDPI_PROTOCOL_CATEGORY_WEB,
        ndpi_build_default_ports(ports_a, 443, 0, 0, 0, 0), ndpi_build_default_ports(ports_b, 0, 0, 0, 0, 0))
        ;
    ndpi_set_proto_defaults (ndpi_str, NDPI_PROTOCOL_ACCEPTABLE, NDPI_PROTOCOL_QUIC,
        1, no_master, no_master, "QUIC", NDPI_PROTOCOL_CATEGORY_WEB,
        ndpi_build_default_ports(ports_a, 0, 0, 0, 0, 0), ndpi_build_default_ports(ports_b, 443, 80, 0, 0, 0)
        );
    ...
}
```

As discussed in section 4.1.1, **ndpi_set_proto_defaults** will add port number to the binary sort tree. It can be seen there are 3 protocols relates to the 443 port. Both **NDPI_PROTOCOL_FBZERO** and **NDPI_PROTOCOL_TLS** with TCP port 443, and **NDPI_PROTOCOL_QUIC** with UDP port 443.

NDPI_PROTOCOL_QUIC protocol can be easily separated from the other two since it is based on UDP protocol. But how to distinguish **NDPI_PROTOCOL_TLS** and **NDPI_PROTOCOL_FBZERO** when receive a TCP packet with port number 443?

Base on our analysis, since the first protocol has register the 443 on the protocol tree. The second protocol will replace the registered node on the binary sort tree. And thus, according to the above code, the corresponding node will record **NDPI_PROTOCOL_TLS** rather than the **NDPI_PROTOCOL_FBZERO**.

Note that it doesn't influent the final guess results. The guessed protocol is just a reference, for a real TLS packet, the callback function **ndpi_search_fbzero** for **NDPI_PROTOCOL_FBZERO** will fail to resolve the packet, but **ndpi_search_tls_wrapper** in the TLS dissector will recognize this packet.

12.2 Question 2: What is the different between the flow's detected_protocol_stack and the packet's detected_protocol_stack?

The packet's **detected_protocol_stack** variable is set in 3 places.

The first place is in the **ndpi_automa_match_string_subprotocol** function, as below:

```
static u_int16_t ndpi_automa_match_string_subprotocol(){
    ...
    if (matching_protocol_id != NDPI_PROTOCOL_UNKNOWN) {
        packet->detected_protocol_stack[1] = master_protocol_id,
        packet->detected_protocol_stack[0] = matching_protocol_id;

        flow->detected_protocol_stack[0] = packet->detected_protocol_stack[0],
        flow->detected_protocol_stack[1] = packet->detected_protocol_stack[1];

        ...
    }
}
```

The second place is in the **ndpi_apply_flow_protocol_to_packet** function, as below:

```
void ndpi_apply_flow_protocol_to_packet(struct ndpi_flow_struct *flow, struct ndpi_packet_struct *
     packet) {
    memcpy(&packet->detected_protocol_stack, &flow->detected_protocol_stack, sizeof(packet->
        detected_protocol_stack));
    memcpy(&packet->protocol_stack_info, &flow->protocol_stack_info, sizeof(packet->
        protocol_stack_info));
}
```

The third place is in the **ndpi_int_change_packet_protocol** function, as below:

```
void ndpi_int_change_packet_protocol (...)  {
    struct ndpi_packet_struct *packet = &flow->packet;

    packet->detected_protocol_stack[0] = upper_detected_protocol;
    packet->detected_protocol_stack[1] = lower_detected_protocol;
}
```

While this function is used only in the **ndpi_int_change_protocol** function, and before calling this function, function **ndpi_int_change_flow_protocol** is called, which set the flow's **detected_protocol_stack** variables, with the same **upper_detected_protocol** and **lower_detected_protocol**.

According to all the analysis, it can be seen that as long as the packet's **detected_protocol_stack** is updated, the flow's **detected_protocol_stack** is updated with same value. Thus, in our opinion, there are no difference between them.

12.3 Question 3: Why using detected_protocol_stack[0] to select the callback function?

Take **check_ndpi_other_flow_func** as an example:

```
u_int32_t check_ndpi_other_flow_func (...)  {
    u_int16_t proto_index = ndpi_str->proto_defaults[flow->guessed_protocol_id].protoIdx;
    int16_t proto_id = ndpi_str->proto_defaults[flow->guessed_protocol_id].protoId;
    NDPI_PROTOCOL_BITMASK detection_bitmask;

    NDPI_SAVE_AS_BITMASK(detection_bitmask, flow->packet.detected_protocol_stack[0]);

    if(...&& NDPI_BITMASK_COMPARE(ndpi_str->callback_buffer[proto_index].detection_bitmask,
        detection_bitmask) != 0 &&...) {
        ...
    }
    ...
}
```

Since **detected_protocol_stack[0]** is corresponding to the application protocol, but **guessed_protocol_id** is corresponding to the master protocol, in general case, **detected_protocol_stack[0]** restricts the **proto_default** array to be selected only when the application protocol equals to the master protocol.

References

[1] Aho, Alfred V.; Corasick, Margaret J. (June 1975). “Efficient string matching: An aid to bibliographic search”. Communications of the ACM. 18 (6): 333340.

[2] Sara Marie Mc Carthy, Arunesh Sinha, Milind Tambe, and Pratyusa Manadhata . (Nov 2016). “Data Exfiltration Detection and Prevention: Virtually Distributed POMDPs for Practically Safer Networks”. Conference: Decision and Game Theory for Security.

[3] T. Dierks. (Aug 2008). “The Transport Layer Security (TLS) Protocol”. RFC5246.