4 Stateful Application Firewall

In this chapter I present a design of a user-space stateful application firewall. This tool was created with an intention to replace existing kernel-based DPI functionality in Endian UTM system due to its complexity of upgrade procedure. In Sections 4.1, 4.2 and 4.3 I describe "three elephants" of the final product: libnetfilter_queue, nDPI and libnetfilter_conntrack. Later on, in Section 4.4 I explain the architecture of the DPI tool I developed and its workflow. There you can find details about logic of the software, flowcharts and code snaps. In the final Section I discuss possible uses of the program and future improvements.

4.1 User-Space Packet Processing With libnetfilter_queue

Libnetfilter_queue (later on NFQ for simplicity) is a user-space Linux library providing an API to manipulate packets that have been queued by the kernel packet filter. It is a library that deprecates old *ip_queue / libipq* system and was previously known as *libnfnetlink_queue* [15]. NFQ library allows to receive queued packets from the kernel nfnetlink_queue subsystem and to issue verdicts on altered packets as well as reinjecting them back into kernel. There are alternatives to using NFQ with better performance like libpcap, but the reason why we chose NFQ was the possibility to selectively process packets e.g. to enqueue only particular traffic into user-space, therefore maintaining high performance. Other libraries read packets directly from NIC, therefore an overhead is created because a program must process even packets that were not intended to be analyzed.

NFQ allows to set the following verdicts on packets:

- NF_DROP – Discard the packet.
- NF_ACCEPT – Let the packet pass and continue traversing iptables' chains.
- NF_QUEUE – Inject the packet into a different queue (the target queue number is in the high 16 bits of the verdict).
- NF_REPEAT – Reinject packet into the same chain once more.
- NF_STOP – Accept packet and stop traversing iptables' chains.

Unfortunately, there is one verdict that is not supported by NFQ library: REJECT. The REJECT target works basically the same as the DROP target, but it also sends back an error message (usually an ICMP packet) to the host sending the packet that was blocked. It is useful for cases when, for example, a user sends an email to a broken server.

When issuing a NF_REPEAT verdict a developer has to make sure that a packet is not re-queued, therefore creating an infinite loop. It can be done by setting a mark on a packet with a special function from the library and enqueuing only those packets which are not marked yet. It can be done by using iptables utility in a following way (just an example):

```
iptables -I INPUT -m mark ! --mark 0x0 -j NFQUEUE --queue-num 1
```

NFQUEUE extension of iptables offers a variety of command line arguments for developing efficient packet processing applications. Indeed, when it comes to reaching maximum performance one would not want to stick to a single-thread application. Fortunately, NFQUEUE offers two options for multi-thread/multi-queue applications:

1. **–queue-balance** – This specifies a range of queues to use. Packets are then balanced across the given queues. This is useful for multicore systems: start multiple instances of the userspace program on queues x, x+1, .. x+n and use "–queue-balance x:x+n". Packets belonging to the same connection are put into the same nfqueue [12].

2. –**queue-cpu-fanout** – This will use the CPU ID as an index to map packets to the queues. The idea is that performance can be improved if there's one queue per CPU. This requires –**queue-balance** option to be specified [12].

A very important feature of NFQ for multi-threaded applications is packet hashing. In cases of flow/packet processing software this feature is very useful. For example, if one wants to implement a functionality to stop processing traffic of a connection that has already seen more than ten packets, it is crucial that packets from the same connection are assigned to the same queue. We utilized this feature when developing a multi-threaded user-space DPI tool described in Section 4.4. Important difference between using "queue balance" option alone and with "cpu fanout" is the way a packet is assigned to a specific queue. When –**queue-balance** is used alone, a hash is computed over packet's source IP, source port, destination IP and destination port; then the packet is assigned to a queue based on the hash value . In this way the library ensures that a every unique network connection is always assigned to the same queue. It is useful for flow processing applications, however, when a single connection is generating high traffic only one queue is overloaded, while others remain idle. In this case –**queue-cpu-fanout** becomes handy, because it changes the packet assignment mechanism to the one based on CPU index for determining the NFQUEUE handling the packet. There is a bunch of other things that can be used for performance increase like increasing a socket buffer size. The whole list can be found in official documentation of libnetfilter_queue [15].

There is no code presented in this section, because there are plenty of examples in the official documentation of libnetfilter_queue [15]. However, I would like to describe what a simplest NFQ-based sniffer written in C should contain. Any C function requires a **main()** method to work, but in our case we will also need a callback function which will be executed for each packet. In this function all the packet analysis, altering, verdict assignment, etc. are done. Main function should be responsible for a library setup, creation of threads and socket error handling. Library setup consists of three steps: initializing NFQ, unbinding existing handler for *AF_INET* (if any exists) and binding *nfnetlink_queue* as *nf_queue* handler for *AF_INET*. Next step would be to start listening on a specified queue number using function called `nfq_create_queue()`. Then, a callback function has to be registered and a packet-receiving loop should be implemented. Furthermore, there are different errors that can appear. The most common one is **ENOBUFS** which indicates that the application cannot process packets fast enough, therefore a program could crash. In this case these errors must be handled by just ignoring packets that caused it or setting a specific socket option to stop receiving these errors at all. The latter can be done by using a `setsockopt()` function to set the **SOL_NETLINK** option to **NETLINK_NO_ENOBUFS** like shown in [28].

4.2 nDPI Deep Packet Inspection Library

nDPI is an open-source cross platform deep packet inspection library based on OpenDpi and maintained by ntop [16]. Currently nDPI supports 234 protocols [17], which include:

- High-level protocols like DNS, FTP, HTTP, POP3, etc.
- Web applications like Facebook, Google, Twitter...
- P2P applications like BitTorrent, Gnutella, eMule, etc.
- Anonymization applications/protocols like Tor and HTTP proxy.
- Many others.

nDPI offers a wide range of features for building effective DPI applications. It can be compiled inside Linux kernel and serve as a kernel-based module of Linux built-in firewall. Also it offers an ability to define port (and port range)-based protocol detection, so that port-independent classification can be complemented with classic port-based detection. Moreover, nDPI has a SSL decoding functionality, therefore a protocol can be figured out using its encryption certificate. In case of applications like Citrix Online and Apple iCloud this is a crucial feature, because otherwise protocols

of these applications would not be detected. Detection process in nDPI is implemented using protocol dissectors and string-based matching. Dissectors are meant to analyze some part of packet's data, like Ethernet header, IP header or the payload. Due to occasional changes in protocols (especially P2P) nDPI is dependent on the community support that provides updates to protocol dissectors.

Unfortunately, the documentation of nDPI is outdated and does not correspond to the implementation of the library. Therefore, below I present the minimal working example of a NFQ firewall with nDPI-based application filtering along with a more detailed description of functions than in the official document [16]. The version of nDPI used in my examples is 2.0.0.

In order to set up nDPI library one should do the following:

Listing 4.1: nDPI library setup

```
NDPI_PROTOCOL_BITMASK bitmask;

set_ndpi_malloc(malloc_wrapper);
set_ndpi_free(free_wrapper);
set_ndpi_flow_malloc(NULL);
set_ndpi_flow_free(NULL);

struct ndpi_detection_module_struct *ndpi_struct =
    ↪ ndpi_init_detection_module();

NDPI_BITMASK_SET_ALL(bitmask);
ndpi_set_protocol_detection_bitmask2(ndpi_struct, &bitmask);
```

First four function calls (lines 3-6) above have purpose of customizing memory allocation and release. As input arguments they accept function pointers and in case it is `NULL` pointer the built-in C functions `malloc()` and `free()` will be used upon calling functions `ndpi_malloc()`, `ndpi_free()` and others. Line 8 is responsible for creation of nDPI detection module. The reason for these functions to exist is because nDPI allows to store custom structures containing flow information, therefore customized memory manipulation functions are needed. Under the hood of `ndpi_init_detection_module()` function different properties are set that influence detection process. For example, there is a possibility to choose which protocol dissectors to use by setting a detection bitmask (last two lines in the example above).

The following code snippet shows how to perform analysis on a packet.

Listing 4.2: nDPI protocol detection preparation

```
struct ndpi_id_struct *src, *dst;
struct ndpi_flow_struct *ndpi_flow = NULL;

ndpi_flow = ndpi_flow_malloc(SIZEOF_FLOW_STRUCT);
memset(ndpi_flow, 0, SIZEOF_FLOW_STRUCT);

src = ndpi_malloc(SIZEOF_ID_STRUCT);
memset(src, 0, SIZEOF_ID_STRUCT);

dst = ndpi_malloc(SIZEOF_ID_STRUCT);
memset(dst, 0, SIZEOF_ID_STRUCT);

u_int64_t tick = ((uint64_t) timestamp.tv_sec) * TICK_RESOLUTION +
timestamp.tv_usec / (1000000 / TICK_RESOLUTION);

struct ndpi_proto detected_protocol = ndpi_detection_process_packet(
    ↪ ndpi_struct, ndpi_flow, packet, packetlen, tick, src, dst);
```

Everything prior to `ndpi_detection_process_packet()` function call is just a preparation of input

arguments for this function. The return value is a structure that contains a parent protocol and a child protocol (master and application protocols in nDPI's terminology). `ndpi_detection_process_packet()` requires the following parameters passed to it for the detection to happen:

1. `struct ndpi_detection_module_struct *ndpi_struct` – The detection structure created in Listing 4.1.
2. `struct ndpi_flow_struct *flow` – Flow void pointer to the connection state machine which is stored in a binary tree.
3. `const unsigned char *packet` – Pointer to the IP header of a packet.
4. `const unsigned short packetlen` – Length of the packet.
5. `const u_int64_t current_tick_l` – Timestamp of a packet.
6. `struct ndpi_id_struct *src` – Pointer to the source subscriber state machine.
7. `struct ndpi_id_struct *dst` – Pointer to the destination subscriber state machine.

You can notice that in the above implementation a piece of memory for *ndpi_flow_struct* structure is allocated and then freed (not included in the listing) for each packet. It leads to a limited detection capabilities due to the fact that TCP and UDP packets can arrive in chunks of data, therefore in case of some protocols/applications detection would be impossible. In order to fix this issue one has to utilize nDPI's implementation of Knuth's algorithm T [54]. The way it can be done is described in detail in Section 4.4, but generally a developer of nDPI-based application has to create a binary tree and write flow data into its leaves, therefore accumulating flow information until one of the dissectors can classify it. In case of TCP a connection is initialized via a TCP handshake which consists of three packets that have to be saved in memory (at least these three): SYN, SYN-ACK and ACK. Moreover, in order to avoid memory overflow (stack or heap) an idle flows' cleanup function has to be implemented. Its role is to traverse the tree and look for flows that have been idle for some period of time and delete them from memory.

At this point combining a NFQ-based packet processor and code from this section in a way that nDPI processes each packet captured by *libnetfilter_queue* should be enough to hack together a simple application layer firewall.

4.3 Building A Stateful Firewall With Linux Connection Tracking Subsystem

Connection tracking in Linux is a subsystem of *netfilter*. Basically, the connection tracking system stores information about the state of a connection in a memory structure that contains the source and destination IP addresses, port number pairs, protocol types, state, and timeout. With this extra information, a more intelligent filtering policies can be defined [38]. Implementation-wise connection tracking in Linux is an in-kernel module that allows stateful packet inspection for iptables. The connection tracking system does not filter packets by itself – they are always passing through, the filtering is done by netfilter.

According to the connection tracking system a connection can be in one of the following five states:

1. NEW – This state means that the connection is valid and has been initialized by a SYN packet, but has not yet been replied to.
2. ESTABLISHED – This state is reached when the firewall has seen a two-way communication.
3. RELATED – The packet is starting a new connection, but it is associated with an existing connection, such as an FTP data transfer [12].
4. INVALID – This state is reached when a packet does not follow the expected behavior of a connection.

5. UNTRACKED – The packet is not tracked at all. The connection must be explicitly set to be untracked in order to reach this state.

Connection tracking module is implemented with a hash table. Each bucket (fast-access location) contains a double linked list of hash tuples. Each connection is represented by two hash tuples: one for original packet direction (who initiated a connection) and the reply direction. Each of hash tuples acts as a container of a tuple which, in its turn represents a relevant information of a connection: source IP, destination IP and layer 4 protocol information. The two hash tuples are embedded in the structure nf_conn [38] which is called a *conntrack* in library's terminology. In Figure 4.1 there is a schematic view of the above information for a clearer picture of this complex data structure.

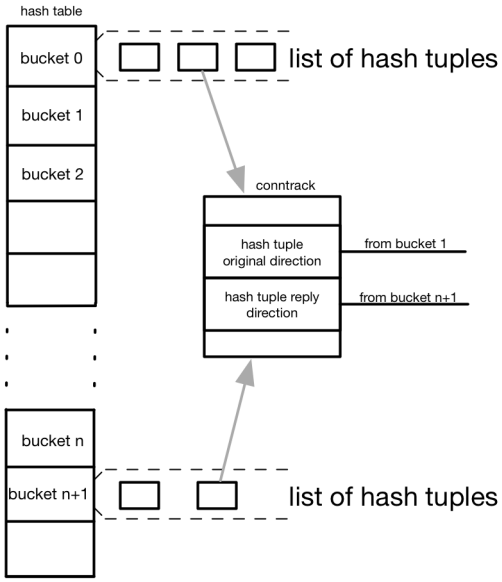


Figure 4.1: Connection Tracking Structure [38]

The above mentioned RELATED state implies that a new incoming connection is somehow associated with an existing one. Determining relationships between connections is a responsibility of concepts called *helpers* and *expectations*. Expectations are connections that are expected to happen in a certain period of time. The concept defined as an **nf_conntrack_expect** structure in the **nf_conntrack_core.h** file [38]. There are application-layer protocols that have certain aspects that are hard to track. The helper searches for patterns that define these aspects and creates profiles of possible connections (expectations) that can happen. For example, in case of FTP protocol the helper looks for the PORT pattern that is sent in reply to the request to begin a passive mode connection (i.e., the PASV method).

From user perspective connection tracking in Linux appears as a set of free software tools that allow system administrators interact, from user-space, with the in-kernel Connection Tracking System described above. These tools are called *conntrack-tools* and consist of two programs [6]:

1. *conntrack* – Command line interface to the connnection tracking system. It can be used to see which connections exist in the hash table and relevant connection information.

2. *conntrackd* – User-space connection tracking daemon that can be used to collect flow-based statistics of the firewall use.

In order to create and manipulate stateful rules using *iptables* program there exist different extensions: *connbytes*, *connlabel*, *connmark*, *connlimit*, etc. [12] The one we are interested in the most is connlabel. This module matches or adds labels to a connection. Labels are bit-based, therefore

all labels may be attached to a flow at the same time. Up to 128 unique labels are supported by default, but there is a possibility of extending it by patching Linux core. Specifically, one should change "XT_CONNLABEL_MAXBIT" value in *include/uapi/linux/netfilter/xt_connlabel.h* file to a higher value.

When it comes to developing flow-based applications netfilter offers a common mechanism to communicate with connection tracking system through user-space. The *libnetfilter_conntrack* library provides an extensive programming interface (API) to the in-kernel connection tracking state table [7]. Main features of this library offer all different sorts of operations on the kernel connection tracking and expectation tables, i.e.: listing, retrieving, inserting, modifying and deleting.

In order to build a stateful DPI tool we decided to use Linux's connection tracking capabilities by using the above mentioned *libnetfilter_conntrack* library and CONNLABEL iptables extension. The idea is to have a predefined set of labels which correspond to the list of protocols supported by nDPI. Then, whenever nDPI has finished detecting a protocol, a label or multiple labels are set for that specific connection. In this way we can later filter connections based on application/protocol they correspond to. Implementation of this functionality should include: library initialization, querying a connection from the hash table and updating labels according to information coming from nDPI library. To query a connection from table one would need five characteristics: protocol (TCP or UDP), source IP, source port, destination IP and destination port. To update connection labels a bitmask has to be created and assigned to the ATTR_CONNLABELS attribute. Completion of the operation can be done with the `nfct_query()` function. The full code example of the above procedure can be found in Listing D.1. An alternative solution is to use a CONNMARK extension of iptables which adds a hexadecimal mark on a connection. It was not chosen due to the fact that Endian was already using this mechanism for other purpose, therefore it would've been confusing to maintain a list of marks for DPI and existing functionality separately.

4.4 Final Architecture And Workflow

Here I present the result of my work with Endian company – an application layer firewall based on the combination of three libraries described above. I will also elaborate on different features of the program explaining the logic behind them. The program was named *NdpiNfqueueFirewall*. Even though in the final implementation it doesn't do any filtering by itself, initially it was an independent user-space firewall with capabilities to block traffic by IP address or port. A decision was made to integrate it with iptables because we didn't want to re-implement part of Linux firewall, thus introducing bugs and maintaining functionality that has been already written.

NdpiNfqueueFirewall is a multithreaded user-space deep packet inspection software created with intention to extend Linux firewall with layer 7 filtering capabilities. It has the following features:

1. Processing packets from multiple queues (one queue per thread).
2. Detection of 227 supported protocols (full list can be found in [18]).
3. Labeling connections for which the protocol detection has ended.
4. Periodic memory cleanup based on time flows have been idle.
5. Fully configurable parameters from the command line.
6. Printing packet header information and protocol to screen.

The following values are configurable from command line:

- Number of queues to listen on, default is 1.
- Number of roots of a binary tree, default is 512.

- Maximum number of flows, default is $2 * 10^8$.
- Time period in milliseconds of scans for idle flows, default is 100ms.
- Maximum amount of time in milliseconds a flow can be idle, default is 30s.
- Maximum number of idle flows, default is 1024.

Memory footprint of the application can be configured by changing number of roots or maximum number of idle flows, because heap memory for these two structures is reserved at start of the program. Normally one should pick a relatively little maximum acceptable number of idle flows and a high frequency of scanning for them, therefore situations when the memory is overfilled with idle connections is unlikely to happen, but memory utilized by the tool remains low.

The implementation of *NdpiNfqueueFirewall* is presented below in a form of flowcharts. For some specific functions I also added code snaps to demonstrate how operations on the binary tree were implemented.

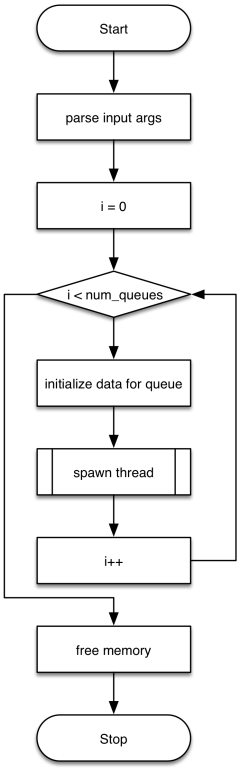
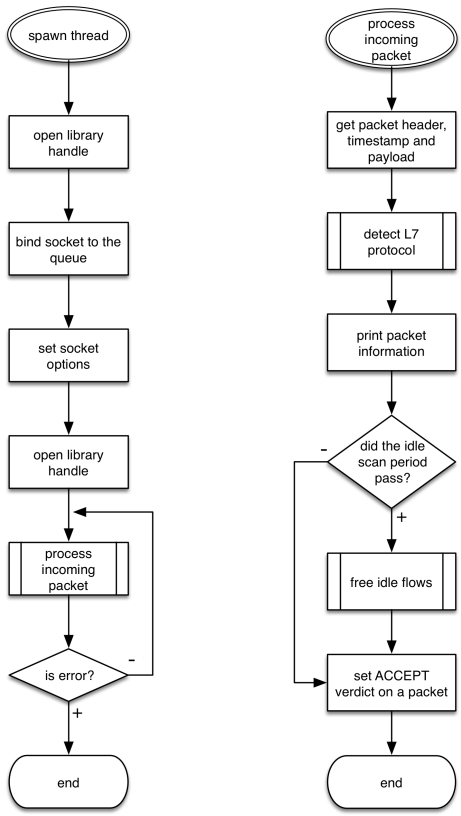


Figure 4.2: Flowchart of the main() function

In Figure 4.2 you can see the flowchart for the main function of the program. It is important to note how threads are spawned. For each queue a custom data structure called *workflow* is created. It contains information that is later used in detection process: binary tree, idle flows, nDPI detection module and other supporting information. In other words, each thread uses its own nDPI instance, therefore errors that usually appear when multiple threads try to write into same block of memory are avoided. This is the reason why it was so important to have packet hashing functionality within NFQUEUE module. The workflow structure is then passed to each thread and a callback functions described below.

Each thread executes NFQ initialization described in Section 4.1 and calls a function that processes each packet (callback function). A decision to create a separate instance of nDPI for each thread was made also because there is no information about thread safety of its APIs. A high-level representation of this process is shown in Figure 4.3a.

Although each thread is bound to a unique queue, the send/recv operations in libnetfilter_queue need to be protected by lock to avoid concurrent writing. Therefore we will apply locking mechanism onto two functions: `nfq_set_verdict()` and `nfq_handle_packet()`. Moreover, multiple socket errors have to be handled when receiving or sending packets from and back to kernel, so different checks are performed to determine a type of an error and appropriate response action.



(a) Spawn thread **(b)** Process packet

Figure 4.3: NFQ initialization (4.3a) and packet processing (4.3b) functions' flowcharts

The callback function mentioned before is where analysis happens. The flowchart of this function can be found in Figure 4.3b. It has a very simple logic and due to smart design of detection functionality flows can be analyzed by calling just one function that corresponds to *"detect L7 protocol"* subprocess block on the flowchart. Callback function also contains code responsible for idle flows' cleanup, but more on that later.

Protocol detection process (Figure E.1a) begins with figuring out whether a packet belongs to an existing flow, it happens in a function represented as *"get flow info"* subprocess block. For this procedure nDPI offers a lot of customization. Firstly, a developer can store any kind of flow information inside the binary tree, in our case we store source IP and port, destination IP and port, hash value, label, ssl information, etc. Full list is available in the source code repository [28] in *ndpi_helper.h* header file. Secondly, a flow comparison function is also customizable and allows developer to perform comparison using any parameter. As you can see in the flowchart (Figure E.1b) the packet's hash value, source and destination information are used to compare it to other flows in the tree. Packet's hash is computed over it's protocol (TCP or UDP), source IP, source port, destination IP and destination port. The way the function works is as follows. First, the tree is traversed until the flow is found or there are no more branches and leaves. In case the flow is found it is returned as is, otherwise a new flow is created, inserted into the tree and a pointer to the new flow is returned.

Back in the detection function (Figure E.1a) analysis continues. Each flow contains a protocol

name and a flag indicating whether detection was completed or not. In case the flow's protocol is known and detection is completed there is nothing else to do in the function, so it exits. Otherwise the `ndpi_detection_process_packet()` function from nDPI library is called which does the protocol detection. After this the above mentioned detection completion flag is set under a specific condition. The condition is the following – the protocol is known or a number of TCP (UDP) packets the related flow has seen is more than 10 (8).

An operation that follows is setting a connection label for further filtering as described in Section 4.3. As it was mentioned before Linux's connection tracking module creates a record in its hash table only when the first packet of a flow traverses all iptables' chains. Because of this reason we cannot set a label on connections whose protocols were discovered since the first packet. In order to handle this particular case we introduced another flag that indicates whether a label was set on a connection or not. Therefore, an attempt to set labels on a connection is made only if the detection process has been completed and a label has not yet been set. In this way we ensure that every connection that has been completely analyzed is labeled. Another trick we use to maximize network performance is enqueuing just enough packets to detect a flow's protocol. We achieve this by reserving a label with ID zero for a special indicator that is set when the detection process is finished. Consequently, when running the program we would just enqueue packets into NFQUEUE only if label with ID 0 is not set i.e. by adding the following to an iptables rule: `-m connlabel ! --label NDPI_DETECTION_OVER`, where NDPI_DETECTION_OVER is a string representation of label 0. This workaround was needed because iptables and CONNLABEL extension do not have a functionality to determine whether there are any labels set on a connection. As a result the maximum number of packets per flow our program analyses is 10 for TCP and 8 for UDP.

The last feature that needs to be mentioned is a memory cleanup functionality. So far we have seen that all new flows are saved in a binary tree, but the program is supposed to run indefinite amount of time, potentially being restarted only together with the OS. For this reason information cannot stay in the binary tree forever because a machine will eventually run out of memory. To avoid this situation we created two functions: `free_idle_flows()` and `node_walker()`. The first function calls internal nDPI function called `ndpi_twalk()` which in its turn recursively calls `node_walker()` for each node to find the idle ones until all nodes of a root have been traversed. As a result of this operation we get a list of pointers to idle flows which are then deleted from the tree in a loop using `ndpi_tdelete()` function. The code is available in Listings D.2 and D.3.

The last version of NdpiNfqFirewall was built for testing purposes using following versions of its dependencies:

1. nDPI v.2.0.0
2. libnetfilter_queue v.1.0.3
3. libnetfilter_conntrack v.1.0.6

4.5 Applications and Future Improvements

Now let's talk about possible applications of the program described above and which advantages libnetfilter_queue gives to it over other libraries. The program was intended to be used on network routers with multi-core CPUs and the primary task was to filter traffic based on application protocols. In order to achieve it a network administrator has to use a correct combination of program's input parameters and iptables rules. For instance, imagine we want to perform deep packet inspection for all routed traffic on a machine with 4 CPU cores. In this case to achieve the best performance NdpiNfqFirewall has to be launched using following command "`./NdpiNfqFirewall -n 4`" which binds each thread to four queues with numbers 10 to 13. Correspondingly, the following firewall rule is needed to correctly enqueue traffic:

```
iptables -I FORWARD -m connlabel ! --label NDPI_DETECTION_OVER -j NFQUEUE --
    ↪ queue-balance 10:13
```

Note: the above command assumes that CONNLABEL configuration file (usually it is located in **/etc/xtables/connlabel.conf**) contains a label named *"NDPI_DETECTION_OVER"* with ID equal to 0. As of other labels, their IDs should be similar to the ones presented in Appendix F.

After this an administrator can do whatever he wants with labeled connections. For example, if Facebook traffic has to be completely blocked the administrator should add the following iptables rule:

```
iptables -I FORWARD -m connlabel --label FACEBOOK -j DROP
```

In a similar fashion traffic coming from any of the supported applications/protocols can be manipulated. Moreover, a network administrator can choose which traffic to analyze (a network, subnetwork, specific host, etc) by enqueuing it. Furthermore, quality of service (QoS) can be done based on application protocols using a combination of a Linux's *tc* tool and iptables. Indeed, this functionality was requested by clients of Endian and was one of the reasons why we decided to combine NdpiNfqFirewall with connection tracking functionality of Linux OS. Except QoS it is also possible to collect network intelligence by reading labels from connection tracking hash table and analyzing data like number of bytes transferred by specific applications or violations of company's network policy (using BitTorrent, p2p protocols, etc.).

NdpiNfqFirewall has a lot of features implemented. However, it is just a proof of concept which requires many improvements to become a production-level program. One of the "must have" characteristics of the production code is automated tests. During my internship I did not have time to fully create a test project for our code, but I made some attempts on writing unit tests. Using a cmocka [4] unit testing framework for C I was able to create a set of mock objects for nDPI functions and write two tests for detection-related functionss. Writing units tests for the nDPI-based application is a very time consuming job, because the idea of a unit test is to validate a single unit of code. Therefore, any function calls and structures from external libraries have to be stubbed or mocked. I created a set of stubs for the following nDPI functions: `ndpi_detection_process_packet()`, `ndpi_set_protocol_detection_bitmask2()`, `NDPI_BITMASK_SET_ALL()`, `ndpi_init_detection_module()`, `set_ndpi_flow_free()`, `set_ndpi_flow_malloc()`, `set_ndpi_flow_free()` and `set_ndpi_malloc()`. These were enough to test the function represented by a flowchart in Figure E.1a, but for other functions more stubs are needed. Moreover, another kind of tests would be a desirable addition to the final product – automated detection tests. Their purpose is to test functionality of the program, in our case it is detection of application protocols. The idea is to have a list of pcap files each of which contains a recording of network activity a specific application. Then this list of files should be sent to the program for analysis and an assertion mechanism will then check which protocols were successfully detected. Currently NdpiNfqFirewall does not support reading packets from files because libnetfilter_queue is lacking this functionality, therefore the program has to be readjusted/extended in order to support automated testing. One way implement it is to create a labeled dataset of network activity of different applications, feed it into the network and then read connection tracking table to compare labels set by NdpiNfqFirewall and those from the dataset.

Another useful improvement would be a disaster recovery feature. A problem appears when one or more threads fail for some reason, in this case a NFQ handler will unbind from corresponding queues and fail. Consequently, packets will stop being queued and program will eventually fail. Instead, in a production environment what should happen is upon failure of one or more threads, reading from queues should be re-established and, ideally, threads should use the same nDPI workflow information. During testing I have not experienced situations when a NFQ handler failed, but all tests were done in an isolated environment, therefore different errors might appear when the program will be run on a "real" network.

Other milestones I have in mind are rather usability improvements than new functionality. In current implementation NdpiNfqFirewall has two usability issues:

1. Inability to set custom queue numbers to bind to.
2. Necessity to maintain a list of labels and their corresponding IDs in *connlabel.conf* file.

The first item above can be easily fixed by creating an additional input argument, so that queue numbers will be set by a user and not be strictly in between 10 and $10+N-1$, where N is the number of queues to listen on. However, the second issue is more complicated to implement. The problem is that nDPI has a list of supported protocols hardcoded into one of its header files, therefore an administrator has to manually create a CONNLABEL configuration file like in Appendix F. It would be nice to implement a functionality that would read a list of protocols supported by nDPI and put them all into **connlabel.conf** file. In this way a user would never have to update his configuration file manually in case of any updates in nDPI code.