Relazione di nDPI_Wrapper
Progetto per il corso di Gestione di Rete A.A 2018/2019

Massimo Puddu (Matricola 531379)

September 13, 2019

Contents

Chapter 1

Introduzione

Questo wrapper serve per poter chiamare le funzioni di nDPI su P ython, in modo da poter semplificare il lavoro ad altri studenti o persone esterne interessate al tool ma che non si vogliono interfacciare direttamente col C.

1.1 Istruzioni per l'esecuzione

spostare la cartella che contiene il progetto nella cartella principale di nDPI. Ora entrare nella cartella del progetto e eseguite

```
. ./automatic.sh
```

Per testare il file d'esempio

```
python3 ndpi_example.py <interface>
```

Per far stampare i risultati si interrompa da tastiera.

1.2 Requisiti e dipendenze

Il software è stato testato su macOS Mojavec utilizzando la versione 3.5 di Python. Per il funzionamento del progetto sono necessari i seguenti moduli:

1. ctypes, dovrebbe essere già incluso nella versione base di Python
2. scapy: pip3 install scapy
   Utilizzato unicamente nel file d'esempio per rendere sequenziale un pacchetto

Chapter 2

Struttura del progetto

Il progetto si suddivide in un file scritto in C e due in python, fa inoltre uso di un Makefile per creare una libreria dinamica che è utilizzata per richiamare alcune funzioni di nDPI in Python. Si è anche fatto uso di uno script bash per automatizzare questa procedura.

2.1 ndpi_wrap.c

Tra gli obbiettivi del progetto avevo quello di far durare nel tempo questo wrapper. Al suo interno sono contenute funzioni create da me per fare da tramite tra le Macro del C e il codice Python. In poche parole al posto di trascrivere direttamente i valori delle Macro nei file ndpi_typestruct.py ho prefito ottenere questi valore dalla fonte originale, pertanto anche se questi valori venissero cambiati non ci sarà bisogno di riscriverli o capire quale di queste è stata modificata.

2.2 ndpi_typestruct.py

È il file più importante in quanto contiene la definizione di tutte le strutture supportate e funzioni che sono pienamente supportate dal progetto. Al suo interno viene introdotta anche la variabile ndpi. Attraverso questa variabile è possibile accedere alle funzioni della libreria di ndpi, infatti in questa variabile viene caricato il file .so creato col Makefile.

2.3 ndpi_example.py

File di esempio in cui si mostra come poter utilizzare il wrapper per poter fare un analisi dei flussi attraverso un interfaccia o un file pcap. Bisogna annotare che ho preferito tenere un file d'esempio il più vicino possibile al C. Un esempio di ciò si ha nella funzione get_flow a riga 115. Ho utilizzato la struttura ad

albero di ndpi andando a chiamare la tfind e tsearch, ma si sarebbe potuto usare una struttura dati qualsiasi di Python che permette di recuperare il flusso. Dopo aver recuperato il flusso relativo al pacchetto. Successivamente si fa una ndpi_detection_process_packet che restituisce la struttura dati ndpi_protocol e andando ad accedere ai dati di questa struttura si riescono a tratte delle conclusioni sul protocollo e se abbiamo ottenuto tutte le informazioni che volevamo da quel flusso o meno.
Una volta analizzati tutti i pacchetti si passa alla funzione result, dove si fanno le seguenti cose:

1. Si fa una ndpi_twalk in cui si va a esplorare l'albero in cui sono memorizzati tutti i flussi e per ognuno di essi si esegue il punto 2

2. Controlla se il protocollo è già stato individuato, e se non lo è si fa una ndpi_detection_giveup. Altrimenti si tengono gli indici dei flussi individuati.

Per finire si usa la ndpi_get_proto_name per ottenere una stringa che rappresenta il procollo.

2.4 Makefile e automatic.sh

Come già spiegato il Makefile è utile per creare la libreria condivisa, mentre più interessante è lo script che si occupa di copiare file utilizzati per wrappare nella cartella /src/lib di ndpi e eseguire in autonomia in Makefile in modo da avere da subito l'ambiente per lavorare.

Chapter 3

API

È possibile chiamare tutte le funzioni di ndpi, ma questo non è del tutto vero in quanto per le funzioni che si vogliono utilizzare bisogna fare prima dei piccoli accorgimenti al interno di ndpi_typestruct.py oppure potrebbe mancare qualche struttura dati che non ho incluso, queste sono le funzione e le stutture che questa versione del wrapper utilizza.

3.1 Struct

Le strutture sono definite all'interno di ndpi_typestruct.py. È molto semplice usare le strutture in quanto per dichiararle è necessario fare come segue:
variabile = nome_struttura().
Per accedere e inizializzare i vari campi: variabile.campo = valore, oppure se la struttura è un puntatore bisogno prima derefernziare per accedere ai campi, questo si ottiene scrivendo: variabile.contents.campo. Per riempire i vari campi alla dichiarazione si può anche optare per questa soluzione: nome_struttura(campo1, campo2, pointer(campo3())). Il terzo parametro è un puntatore e si crea a partire da una struttura esistente, questo valore non sarà mai NULL di conseguenza, se si vuole ottente quel valore bisogna inizializzare il terzo paramentro come POINTER(nome_struttura)().
Di seguito illustrerò le strutture che ritengo più importanti.

```
#struttura che definisce quali protocolli devono essere trovati, è consigliabile
#non inizializzarla si python ma ottenerne un istanza tramite la funzione
#ndpi_init_detection_module() vedere sezione 3.2.
#se si interessa settare solo determinati protocolli è consigliato andare a vedere
#il codice di ndpi_main.c riga 2825(inizio funzione ndpi_set_protocol_detection_bitmask2)
class ndpi_detection_module_struct(Structure)
#struttura che contiene i campi relativi a un determinato flusso, e
#che aiuta l'identificazione dello stesso.
class ndpi_flow_struct(structure)
#si tratta della struct ndpi_iphdr con byte order little endian si può accedere a
```

```
#tutti i campi di un pacchetto ip con la solita sintassi es. variabile.tos
class struct_ndpi_iphdr_little_end(structure)
#contiene 3 campi ognuno utile per capire con che protocollo si lavora
#master_protocol es. HTTP
#app_protocol es. Facebook
#category è un ndpi_protocol_category_t perchè l'utente può definire propri protocollo
class ndpi_protocol(structure)
```

3.2 Funzioni

Tutte le seguenti funzioni possono essere chiamate nel seguente modo ndpi.nome_funzione(param). Quelle che descriverò qui sotto sono i nomi delle funzioni ufficialmente supportate, si prega di fare particolare attenzione alla funzione ndpi_detection_process_packet. Per aumentarle si prega di leggere il capitolo 4

```
#ritorna un c_void_p dovrà essere castato in c_char_p, contiene la versione di ndpi.
ndpi_revision()
#ritorna un intero che rappresenta la versione dell'api
ndpi_wrap_get_api_version()
#ritorna un intero che viene utilizzato indicare la dimensione di alcuni array
ndpi_wrap_ndpi_num_fds_bits()
#ritorna un intero che rappresenta il numero massimo delle custom categories
ndpi_wrap_num_custom_categories()
#ritorna un intero che rappresenta la lunghezza dei nomi delle custom categories
ndpi_wrap_custom_category_label_len()
#ritorna un intero che specifica il numero massimo di protocolli supportati
ndpi_wrap_ndpi_max_supported_protocols()
#ritorna un intero che specifica il numero massimo di custom categories
ndpi_wrap_ndpi_max_num_custom_protocols()
#ritorna un intero che rappresenta la dimensione di un protocollo
ndpi_wrap_ndpi_procol_size()
#funzione che serve per inizializzare a 0xFFF tutti i protocolli della bitmask
ndpi_wrap_NDPI_BITMASK_SET_ALL(NDPI_PROTOCOL_BITMASK* bitmask)
#ritorna None se non trova key, restituisce key altrimenti
#key è di tipo c_void_p(se si utilizza un altra struttura
#è consigliabile un cast) è l'elemento da ricercare nell'albero
#root è di tipo c_void_p e rappresenta l'albero nel quale sono memorizzate le key
#funzione che confronta key con un elemento dell'albero, è presente un esempio
#nella riga 58 di ndpi_example.py e segue la riga 79
ndpi_tfind(key, root, fun)
#ritorna la key presente nell'albero, se non è presente la aggiunge.
#key come prima
#root è di tipo pointer(c_void_p) e si utilizza lo stesso root precedente
#fun come prima
ndpi_tsearch(key, root, fun)
#esplora l'albero applicano la funzione fun su ogni nodo
```

```
#root è la radice dell'albero
#fun deve avere come parametri il nodo, un intero che specifica quando è stato chiamato,
#la profondità e l'user data, vedi la funzione a riga 47 per un esempio
ndpi.ndpi_twalk(root, fun, user_data)
#funzione che chiama la funzione foo su ogni nodo dell'albero prima di
#cancellare il nodo root albero da eliminare
#fun è la funzione da chiamare prima di eliminare effettivamente il nodo
ndpi_tdestroy(root, fun)
#ritorna un nuovo ndpi_detection_module_struct inizializzato
ndpi_init_detection_module()
#ritorna None e setta la bitmap di ndpi_struct per capire quali protocolli identificare
#ndpi_struct è un ndpi_detection_module_struct
#la bitmask è un NDPI_PROTOCOL_BITMASK
ndpi_set_protocol_detection_bitmask2(ndpi_struct, bitmask)
#ritorna un ndpi_protocol, impostandone un ID.
#ndpi_struct è il modulo di identificazione
#il flusso da dare al modulo di identificazione
#ATTENZIONE il campo packet deve essere presentato come una sequenza continua di dati
#e deve contene sia il pacchetto a lv.3 che a lv.4 per questa ragione ho usato scapy
#in quanto con pyshark avrei dovuto ricostruire il pacchetto, es. a riga 175
#lunghezza del pacchetto
#timestamp del pacchetto
ndpi_detection_process_packet(ndpi_struct, flow, packet, packetlen, current_tick, src, dst)
#ritorna un ndpi_protocol, funzione da chiamare prima di rinunciare alla
#identificazione di un flusso, per ridurre gli NDPI_UNKNOWN_PROTOCOL
#ndpi_struct è il modulo di identificazione
#il flusso da dare al modulo di identificazione
#enable guess se è unknown
ndpi_detection_giveup(ndpi_struct, flow, enable_guess)
#ritorna un c_void_p dovrà essere castato in c_char_p, contiene in nome del
#protocollo rappresentato da proto_id.
#ndpi_mod è un pointer(ndpi_detection_module_struct), e contiene i protocolli supportati
#proto_id è un intero che rappresenta il protocollo da tradurre
ndpi_get_proto_name(ndpi_mod, proto_id)
```

Chapter 4

Espansioni future

Ho cercato di ricoprire per quanto possibile le funzioni e strutture principali ma ovviamente qualcuno in futuro potrebbe aver bisogno di strutture o funzioni che non ho incluso nel progetto. Per poterle aggiungere è sufficiente modificare il file ndpi_typestruct.py.

4.1 Strutture

la dichiarazione avviene nel seguente modo

```
class nome_struttura(Structure):
```

da qui si può:

1. Se la struttura usa un puntatore di se stessa o una struttura che dovrà essere devinita successivamente si dovrà usare il comando pass e quando sarà tutto definito utilizzare il comando per riempire la struttura

```
nome_struttura._fields_ = [campi]
```

2. Se la prima condizione non è rispettata si possono definire immediatamente i campi in maniera molto simile

```
class nome_struttura(Structure):
    _fields_ =[campi]
```

Per quanto riguarda i campi prendiamo un esempio. se abbiamo in C:
"u_int8_t nome:1"
questo all'interno delle quadre precedenti deve diventare
"("nome", c_uint8, 1)"

4.2 Funzioni

Le funzioni sono più facili da sbrigare in quanto ci interessanto principalmente due aspetti, il restype e argtypes. Il riempimento di questi due campi rende le funzioni supportate(non sempre vero, in casi semplici sarà ctypes a gestire il tutto). Degli esempi di ciò si trovano alla fine del file ndpi_typestruct.py. Un rapido esempio è
ndpi.nome_fun.restype = valore di ritorno della funzione C
e argtypes
ndpi.nome_fun.argtypes = [tipi dei paramentri della funzione]

4.3 Miglioramenti

Partendo dalle piccole cose si potrebbe iniziare spostando le funzione supportate in un file specifico a parte, in modo da differenziare meglio le strutture dalle funzioni e rende il tutto più leggibile, si potrebbero spostare inolte le seguenti variabili create nel file d'esempio in modo da non farle definire sempre all'utente: CMCFUN,GUESS e FREE.
Se si vuole rendere invece questo wrapper il più lontano possibile dal C si potrebbe iniziare a nascondere tutto il tipaggio. Facendo questo però bisogna andare a creare delle funzioni supplementari per il controllo, l'utente non dovrà andare andare a chiamare le funzioni contenute nella variabile ndpi ma potrà chiamare solo funzioni definite nel nostro wrapper che poi andranno a chiamare quelle native di C, ovviamente non dovrà essere l'utente a controllare il tipo. Inoltre si potrebbero creare altre funzioni che inizializzano in automatico delle strutture(che utilizzano pacchetti) in modo da far avere all'utente un esperienza più vicina possibile al Python e lontana dal C. Si può inoltre pensare di permettere l'utilizzo di pyshark in questo caso sarà necessario ricostruire il pacchetto in mado che diventi una sequenza continua di bytes.

Chapter 5

Conclusioni

Spero di aver realizzato un wrapper il più utile possibile ad altri studenti e terzi in modo da poterli aiutare nei loro progetti.