# Sniffare il traffico di Rete con Python

Un Caso di Studio

Giorgio Zoppi <zoppi@atsystem.org>

# Note su Python

- Linguaggio interpretato a macchina virtuale
  - Scoping statico
  - Passaggio di parametri per valore
  - Macchina Virtuale implementata in : C, C#, Java
  - Ottimo per le ricerche su stringhe

Metafora: Sviluppo rapido, sviluppo meglio.

- http://www.python.it/

# Python e lo Sniffing

- Esistono i wrapper python per le comuni librerie C
  - pcap, dnet, nids …
- Compromesso: performance / tempo / costi
- Moduli, alcuni sono incompleti
  - pypcap/pcappy – pcap per python
  - dpkt – libreria di ricostruzione pacchetti
  - pynids – wrapper per le libnids

# Cosa faremo :)

- Esempio di sniffing con le pcap: contare i pacchetti IP
- Un esempio di ngrep sul payload dei pacchetti
- Un esempio con le libnids usando il pattern matching multiplo nel traffico smtp segnando le mail come spam

# *Exact set matching problem*

Localizzare occorrenze di più pattern appartenenti ad un insieme $P = \{P_1, \dots, P_k\}$ in un testo T[1,...,m]

Sia n = $\sum_{i=1}^{k} |P_i|$

**Exact set matching** può essere risolto con un tempo $O(|P_1| + m + \dots + |P_k| + m) = O(n + km)$

per applicare ogni exact matching di tipo lineare k volte

**Algoritmo Aho-Corasick (AC)**

- Classica soluzione di exact set matching
- Lavora in $O(n + m + z)$, dove z è il numero di pattern occorrenze in T

# *Keyword Trees*

Un **keyword tree**(o **trie**) per un insieme di pattern *P*
è un albero *K* con le seguenti caratteristiche:

- ogni arco è etichettato con un carattere
- due archi uscenti di un nodo hanno etichette differenti

Si definisce l'**etichetta di un nodo** *v* come la concatenazione delle etichette sugli archi che compongono il cammino dal nodo radice a *v*, e si indica con ***L(v)***.

- per ogni P ϵ *P* c'è un nodo *v* con *L(v)*= P
- e l'etichetta *L(v)* di ogni foglia *v* equivale a qualche P ϵ *P*

# *Keyword Tree: Costruzione*

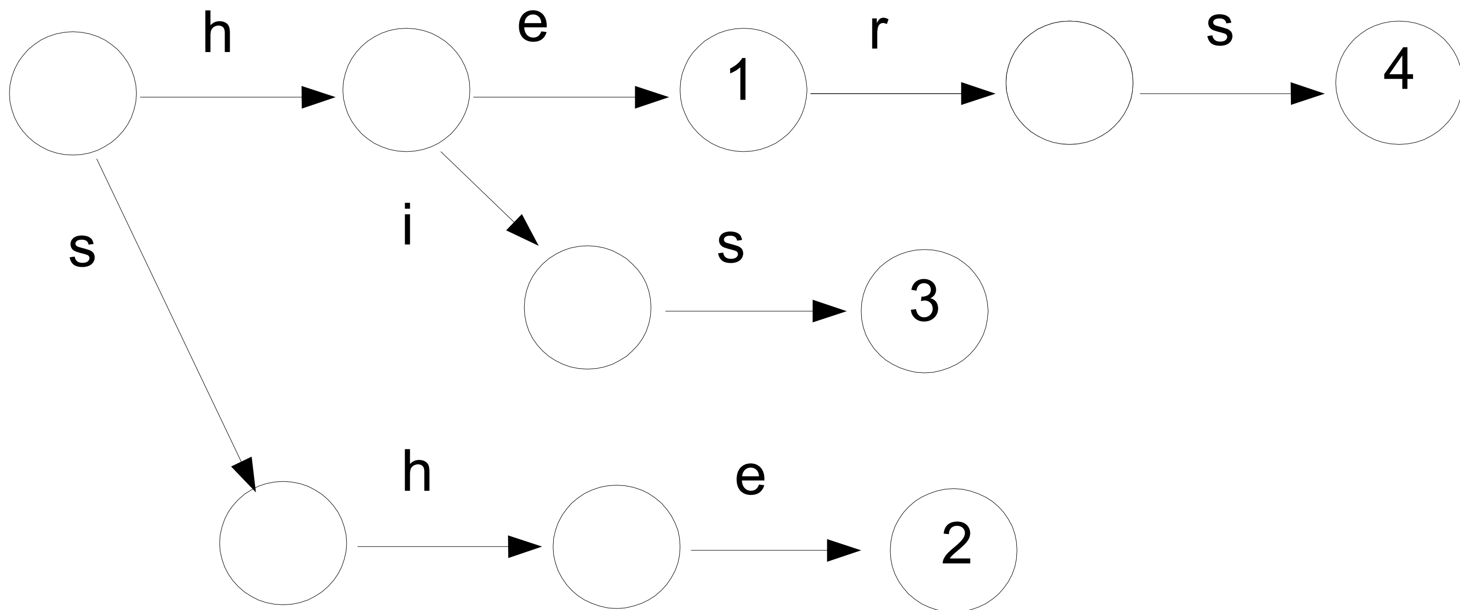
**Costruire** un keyword tree per $P = \{P_1,\ldots,P_k\}$

- Iniziare dal nodo radice
- Inserire ogni pattern $P_i$, uno dopo l'altro nel seguente modo:
  - Partire dal nodo radice, seguendo i percorsi etichettati con i caratteri di $P_i$. Se il percorso finisce prima di $P_i$, continuarlo aggiungendo un nuovo arco e nodo per i rimanenti caratteri di $P_i$.
  - Salvare l'identificatore *i* di $P_i$ nell'ultimo nodo del percorso.

  Complessità: $O(|P_1| + \ldots + |P_k|) = O(n)$

# *Esempio di un Keyword Tree*

Il keyword tree per $P = \{he_1, she_2, his_3, hers_4\}$



# Keyword Tree: Ricerca

La **ricerca** di una stringa P:

Partire dal nodo radice e seguire il cammino etichettato da caratteri di P fino a quando è possibile.

•Se il cammino si ferma in un nodo in cui è presente un identificatore allora P è una chiave del dizionario;

•Altrimenti se termina prima di P, la stringa non appartiene al dizionario.

Complessità: $O(|P|)$

Un'applicazione banale al pattern matching dovrebbe portare a $O(nm)$.

Estenderemo un keyword tree ad un automa per realizzare il matching in tempo lineare

# *Automa di Aho-Corasick(1)*

**Stati:** nodi del keyword tree (trie).
**Stato iniziale: 0 = la radice.**

L'automa viene definito mediante tre funzioni:

1. la funzione **goto** *g(q,a)* da lo stato successivo partendo dallo stato q facendo il matching del carattere a (funzione di transizione).
   Se un arco(q,v) è etichettato da a, allora g(q,a) = v;
   g(0,a) = 0 per ogni a che non etichetta un arco fuori dalla radice.
   L'automa rimane nello stato iniziale se scandisce caratteri che non corrispondono.

Altrimenti g(q,a) = 0

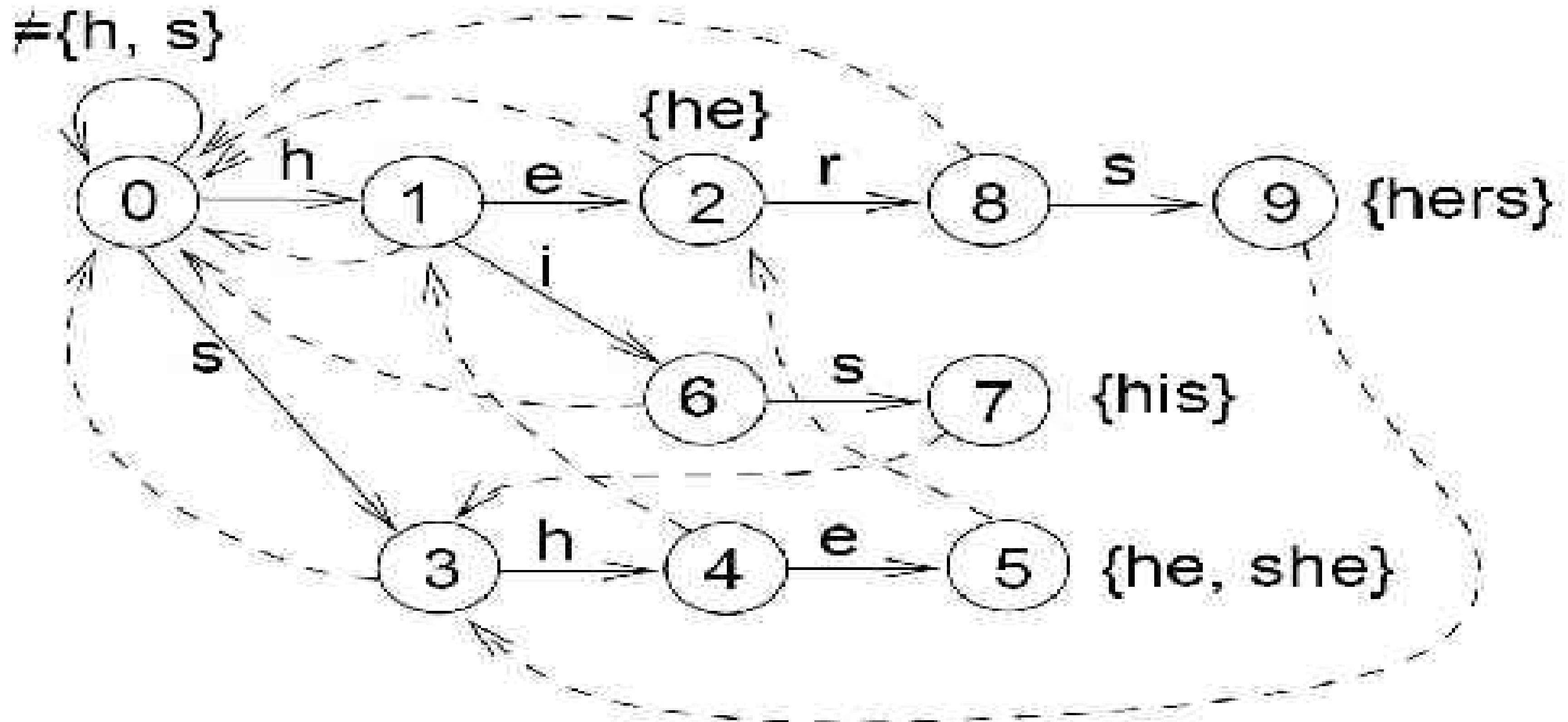
# *Automa di Aho-Corasick(2)*

2. la **funzione failure** *f(q)*, per $q \neq 0$ dà lo stato in cui mi trovo quando ho mismatch

*f(q)* è il nodo etichettato dal *più lungo suffisso proprio w* di *L (q)* tale che *w* sia prefisso di qualche pattern => una transizione di fallimento non manca nessuna potenziale occorrenza. Notiamo che f(q) è sempre definita, poichè L(0) = ε è un prefisso di ogni pattern.

3. La **funzione output** *out(q)* ci dà l'insieme dei patterm riconosciuti quando entriamo in uno stato *q*.

## *Esempio di un Automa AC*




Le linee tratteggiate sono le transizioni di fallimento

# *Ricerca AC nel testo T[1..m]*

```
# Pseudocodice  Python.

q = 0  # stato iniziale root
# range(m+1) funzione che restituisce
#  interi da 1 a m.
for i in range(1,m+1):
 while (not Empty(g(q,T[i]))):
    q =f(q); # segui un fallimento
  q = g(q,T[i]);# segui un goto
  if (not Empty(out(q))):
    print i,out(q)
```

# Primo esempio con le pycap

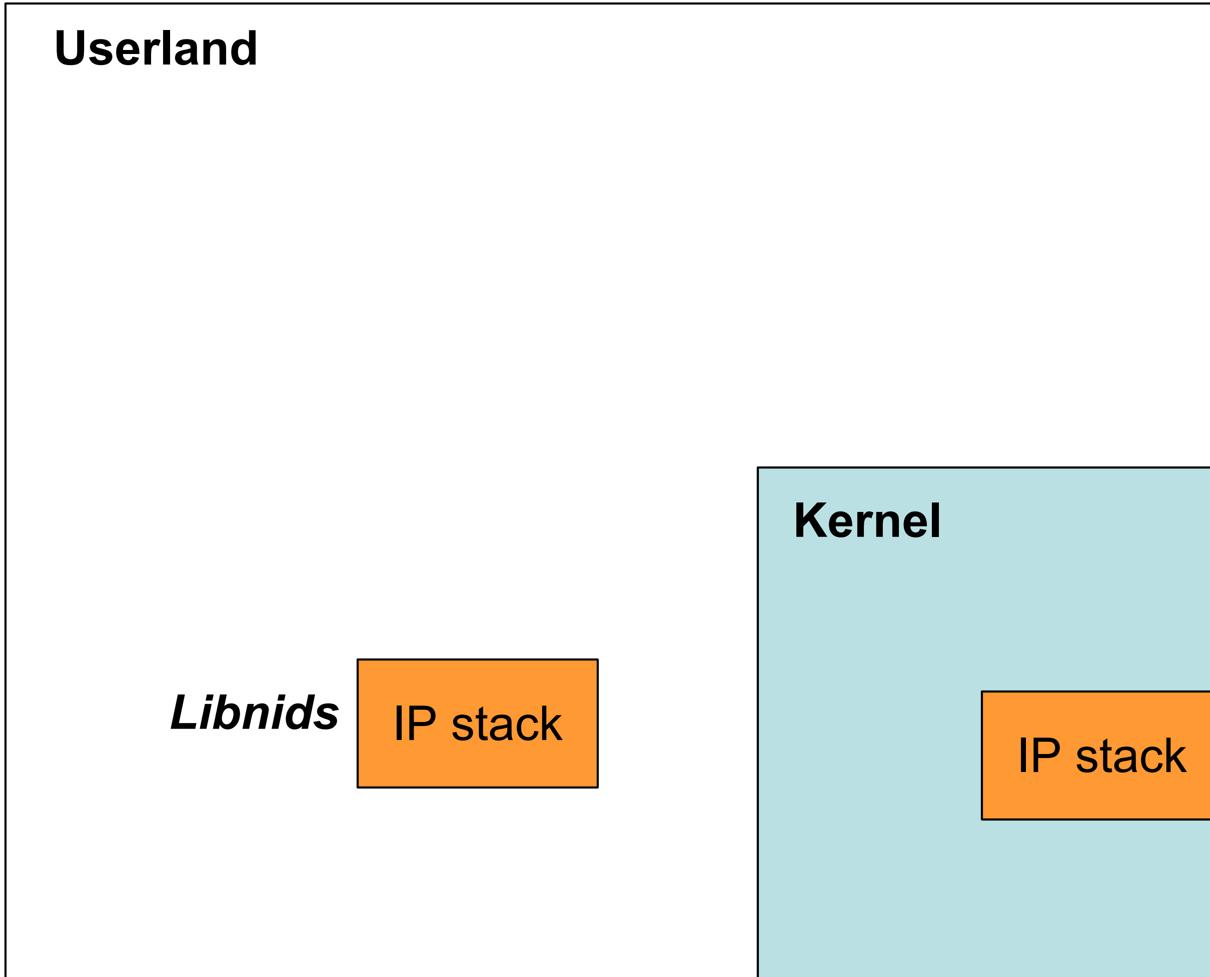
# libnids – riassembla gli stream IP

NIDS “E” box (event generation box)
Userland TCP/IP stack
Rimarca lo stack IP  di Linux 2.0.36 IP internamente
Si basa su libnet e libpcap
IP fragment reassembly

Userland

Kernel

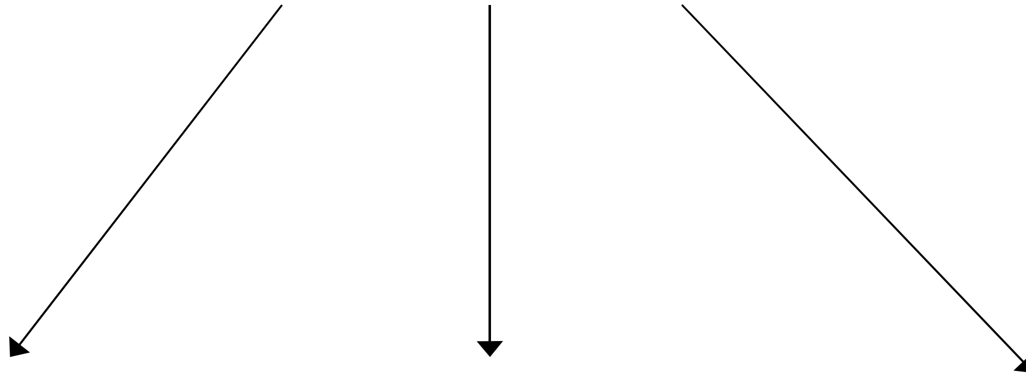
IP stack


Userland
Kernel
Libnids
IP stack
IP stack

# Fasi di Libnids

- Inizializzazione
  - **`nids_init()`**
- Registrazione callback
  - **`nids_register_tcp()`**
  - **`nids_regster_ip()`**
  - **`nids_regiser_udp()`**
- Esecuzione
  - **`nids_run()`**
- Reazione
  - **`nids_kill_tcp()`**

# `nids_run()`



**TCP callback**

TCP stream object:
- TCP state
- client data
- server data
- source IP, port
- dest IP, port
- seq, ack, etc …

**UDP callback**

UDP packet:
- source IP, port
- dest IP, port
- UDP payload

**IP callback**

IP packet
- struct IP packet

# Automa di libnids

- **`NIDS_JUST_ESTABLISHED`**
  - L'handshake TCP ha avuto luogo
  - Devi impostare **`stream->{client,server}.collect=1`** per ottenere i payload raccolti
- **`NIDS_DATA`**
  - Ci sono dati e la connessione è stata stabilita
- **`NIDS_RESET`, `NIDS_CLOSE`, `NIDS_TIMED_OUT`**
  - La connessione TCP è stata persa, resettata o si è verificato un timeout

# Introduzione a pynids

- Interfaccia event driven (nids_run(), nids_next())
  - TCP stream reassembly
  - espone lo stato TCP
  - Crea un oggetto TCP
- Mantiene gli indirizzi, i dati, ecc
  - Riassembly TCP

# Passi di base per Libnids

- Inizializzazione
  - `nids_init()`
- Stabiliamo un parametro
  - `nids.param(“`*`attribute`*`”, value)`
- Registriamo le callback
  - `nids.register_tcp(handleTcp)`
  - `def handleTcp(tcp): …`
- Esecuzione!
  - `nids_run()`
  - `while 1: nids_next()`

# Elaborazione locale in Pynids

- Nota: Ho registrato le callback.
- Durante il transito dei pacchetti cosa succede?

- TCP
  - Posso creare nuovo stato interno o riusare
  - Append data
  - L'elaborazione secondo lo stato che vede la callback
- UDP o IP?
  - L'elaborazione (stateless) avviene nella callback

# Esempio di codice (Python)

```
import nids
<handleTcpStream>

def main():
    nids.param("scan_num_hosts", 0)
    if not nids.init():
        print "error -", nids.errbuf()
        sys.exit(1)
    nids.register_tcp(handleTcpStream)
    try:        nids.run()  # loop forever
    except KeyboardInterrupt:
        sys.exit(1)
```

# Esempio di codice Python

```
def handleTcpStream(tcp):
   if tcp.nids_state == nids.NIDS_JUST_EST:
       if dport in (80, 8000, 8080):
           tcp.client.collect = 1
           tcp.server.collect = 1
    elif tcp.nids_state == nids.NIDS_DATA:
        tcp.discard(0)
    elif tcp.nids_state in end_states:
        print "addr:", tcp.addr
        # may be binary
        print "To server:‟, tcp.server.data
        print "To client:‟, tcp.client.data
```

# Esempio di codice in C

```
int main(int argv, char *argv[])
{
    if (nids_init() == 0)
        err(1, “error, %s”, nids_errbuf);
    nids_register_tcp(handleTcp);
    nids_run();
    exit(0);
}
```

# Esempio di codice in C

```
int handleTcp(struct tcp_stream *tcp)
{
    switch (tcp->nids_state) {
    case 'NIDS_JUST_EST':
        if ((tcp->addr.dest == 80) ||
            (tcp->addr.dest == 8000) ||
            (tcp->addr.dest == 8080) {
            tcp.server.collect = 1;
            tcp.client.collect = 1;
         }
         break;
     case 'NIDS_DATA':
            nids_discard(tcp, 0);
            break;
     case 'NIDS_CLOSE':
     case 'NIDS_RESET':
     case 'NIDS_TIMED_OUT':
         printf("((%s, %d), (%s, %d))\n", inet_ntoa(tcp->saddr), tcp.srce,

                inet_ntoa(tcp->daddr), tcp.dest);
         printf("%s\n", tcp->server.data);
         printf("%s\n", tcp->client.data);
         break;
    }
}
```

**Quasi le stesse LoC prima di manipolare le stringhe**

# VersionDetect

- Semplice programma Python scritto da Nazario
- Rileva la versione di diversi protocolli
  - Supporto per : SSH (client, server), WWW (client, server), e SMTP
- Motivazione: raccogliere dati per fare stack fingerprinting

63.236.16.161 SymbianOS 6048 (on Nokia 7650?) www 80/tcp
63.236.16.161: 80: Microsoft-IIS/6.0

# Output di VersionDetect

```
   192.168.1.7:  22:  SSH-2.0-OpenSSH_3.5
 192.168.1.101:http:  Mozilla/5.0 (X11; U; OpenBSD i386; en-
         US; rv:1.5a) Gecko/20031030 Mozilla Firebird/0.6.1
  168.75.65.85:  80:  Microsoft-IIS/5.0
   165.1.76.60:  80:  Netscape-Enterprise/3.6 SP2
  168.75.65.69:  80:  Microsoft-IIS/5.0
  168.75.65.87:  80:  Microsoft-IIS/5.0
   69.28.159.7:  80:  ZEDO 3G
 198.65.148.234:  80:  Apache/1.3.29 (Unix) PHP/4.3.3
216.150.209.231:  80:  Apache/1.3.31 (Unix)
 212.187.153.30:  80:  Apache/1.3.31 (Unix)
 212.187.153.37:  80:  Apache/1.3.31 (Unix)
 212.187.153.32:  80:  thttpd/2.25b 29dec2003
 64.209.232.207:  80:  Apache/1.3.27 (Unix)
                       mod_perl/1.27
  216.239.39.99:  80:  CAFE/1.0
```

# Controllare dati con  pynids

- Nota: tcp.{server, client}.data sono solo stringhe
- Allora: funzionera ogni manipolazione di stringhe
  - Ricerca

```
if “HTTP/1.0” in tcp.client.data:
```

  - Ricerca mediante espressioni regolari

```
if re.search(“HTTP/1.[10]”, tcp.client.data):
```

  - Sostituzione di parti di stringhe

```
string.replace(req, “GET HTTP/1.0”, “”, 1)
```

# Risorse

- http://www.tcpdump.org/
- http://www.packetfactory.net/projects/libnids/
- http://monkey.org/~provos/libevent/
- http://monkey.org/~dugsong/{dpkt, pycap}
- http://oss.coresecurity.com/projects/pcapy.html
- http://monkey.org/~jose/software/flowgrep/
- http://pilcrow.madison.wi.us/pynids/

# Thanks

Antonio Cisternino <cisterni@di.unipi.it>

# Thanks