Chương trình KC-01:
Nghiên cứu khoa học phát triển công nghệ thông tin và truyền thông

Đề tài KC-01-01:
Nghiên cứu một số vấn đề bảo mật và an toàn thông tin cho các mạng dùng giao thức liên mạng máy tính IP

Báo cáo kết quả nghiên cứu

PHẦN MỀM BẢO MẬT MẠNG DÙNG GIAO THỨC IP

Quyển 4A: "Các phần mềm bảo mật gói IP trên hệ điều hành Linux"

HÀ NỘI-2002

Báo cáo kết quả nghiên cứu

PHẦN MỀM BẢO MẬT MẠNG DÙNG GIAO THỨC IP

Quyển 4A: "Các phần mềm bảo mật gói IP
trên hệ điều hành Linux"

Chủ trì nhóm thực hiện:
TS. Trần Duy Lai

MỤC LỤC

PHẦN I
LẬP TRÌNH MẠNG TRONG LINUX

3. Các thủ tục hỗ trợ mức cao hơn
4. Thiết bị mạng
4.1 Cấu trúc cơ bản của một thiết bị mạng
4.2 Đặt tên cho thiết bị mạng
4.3 Đăng ký một thiết bị
4.4 Cấu trúc thiết bị
4.5 Hàng đợi
5. Các hàm (methods) thiết bị mạng
5.1 Setup
5.2 Truyền frame
5.3 Frame Headers
5.4 Reception (nhận)
6. Các hàm tùy chọn (optional functionality)
6.1 Activation và Shutdown (Kích hoạt và tắt)
6.2 Configuration và Statistics
7. Multicasting
8. Các thủ tục hỗ trợ Ethernet

PHẦN II
CÁC SẢN PHẨM BẢO MẬT GÓI IP

A. PHẦN MỀM TRANSCRYPT

1. Chương 1-Giải pháp Transcrypt **2.**
3. Tổng quan
1.1 Các tầng mạng và mã hoá
1.2 Định tuyến IP và mạng riêng ảo (Virtual Private Network)
1.3 Cách làm việc của Transcrypt
1.4 Các thành phần của TRANSCRYPT
1.5 Mã nguồn

4. Mô tả giao thức
2.1 Mã hoá các gói
2.2 Trao đổi khoá
2.3 Các vấn đề về bảo mật
3.Giải pháp can thiệp mật mã trong Transcrypt
3.1 Gói tin được gửi đi
3.2 Nhận gói tin

Chương 2-Phần mềm TRANSCRYPT
1. Quá trình cài đặt
1.1 Điều kiện tiên quyết
1.2 Mã nguồn của TRANSCRYPT
1.3 Biên dịch
1.4 Cài đặt chương trình TRANSCRYPT
1.5 Thiết lập cấu hình
1.6 Chạy chương trình TRANSCRYPT
2. Cấu hình phần mềm TRANSCRYPT
2.1 Các tuỳ chọn đặc biệt
2.2 Danh sách tất cả các tham số
2.3 Lỗi điều khiển trong '*transcryptd*'
3. Cài đặt TRANSCRYPT trên mô hình thử nghiệm
3.1 Cấu hình mạng
3.2 Thiết lập các file cấu hình để thực hiện kết nối TRANSCRYPT

B. PHẦN MỀM IP-CRYPTO

Chương 1- Giải pháp bảo mật của IP-CRYPTO
1. Quản lý các gói tin mạng trong nhân Linux
2. Kỹ thuật tạo card mạng ảo và cách gửi gói tin qua card mạng ảo
3. Nhận gói tin mạng trong nhân Linux
4. Các mô hình bảo mật gói tin ở tầng IP trong IP-Crypto
4.1 Mô hình hoạt động với sự tạo lập đường hầm trong IP-Crypto (tunnel mode)
4.2 Bảo mật ở tầng IP cho phiên truyền thông giữa hai máy (Transport mode)
4.3 Định dạng gói tin đóng viên ESP (Encapsulating Security Payload Packet Format)
5. Quá trình gửi gói tin trong IP-Crypto
6. Nhận gói tin trong IP-Crypto

Chương 2-Phần mềm IP-CRYPTO
1. Mã nguồn của IP-CRYPTO
2. Quá trình biên dịch và cài đặt IP-CRYPTO
2.1 Các yêu cầu

2.2 Cài đặt ở chế độ kernel
2.3 Cài đặt ở chế độ module
3. Thiết lập cấu hình cho IP-CRYPTO
3.1 Cấu hình mạng
3.2 Cấu hình IP-CRYPTO
4. Các tệp sau khi cài đặt

C. PHẦN MỀM DL-CRYPTOR

Chương 1-Bảo mật ở tầng DataLink
1. Cấu trúc gói tin MAC (Medium Access Control)
2. Lập trình module bảo mật mạng
2.1 Lập trình module
2.2 Cài đặt và xoá bỏ module
2.3 Module bảo mật mạng ở tầng datalink

Chương 2-Phần mềm DL-Cryptor
1. Mã nguồn DL-Cryptor
2. Quá trình biên dịch và cài đặt DL-Cryptor
2.1 Các yêu cầu
2.2 Dịch nhân mới
2.3 Biên dịch module datalink
3. Thiết lập cấu hình cho DL-Cryptor
4. DL-Cryptor và trao đổi khóa tự động

D. GIẢI PHÁP MẬT MÃ

Chương 1-Mã dữ liệu bằng mã khối
1. Mã khối MK1
1.1 Giới thiệu chung về mã khối
1.2 Các cấu trúc mã khối cơ bản
1.3 Nghiên cứu về các mô hình mã khối an toàn-chứng minh được
1.4 Mô hình mã khối MK-1
2. MK1 trong các phần mềm bảo mật gói IP
2.1 Mode hoạt động của mã khối MK1
2.2 Khoá dùng trong MK1

Chương 2-Trao đổi khoá tự động
1. Thủ tục trao đổi khoá có xác thực
2. Định nghĩa của thủ tục an toàn
3. Các đặc trưng mà thủ tục cần có
4. Thủ tục STS (trạm-tới-trạm)
5. Chương trình kex (key exchange) – Version 1.0
6. Sử dụng chương trình KEX

7. Các đặc tính của KEX
8. Trao đổi khoá tự động trong TransCrypt
9. Trao đổi khoá tự động trong IP-Crypto
10. Trao đổi khoá tự động trong DL-Cryptor

PHẦN I

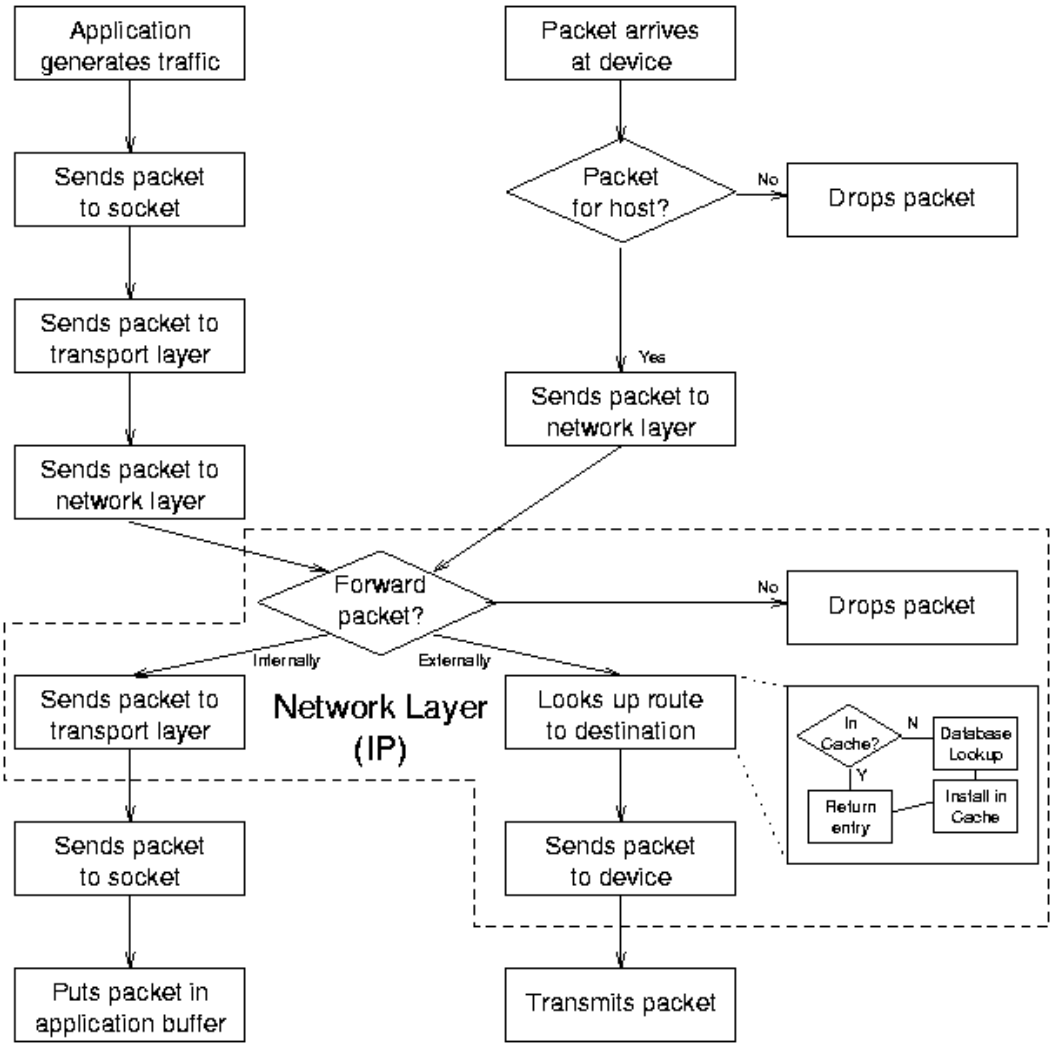
LẬP TRÌNH MẠNG TRONG LINUX

Chương 1
Mạng IP trong Linux

1. Tổng quan về truyền thông thông điệp

Mục này sẽ trình bày tổng quan về toàn bộ hệ thống truyền tin mạng trong Linux. Nó cung cấp một số thảo luận về các cách cấu hình, các cấu trúc dữ liệu và mô tả các cơ sở của việc dẫn đường IP.

1.1 Đường dẫn truyền thông mạng



Hình 1: Truyền thông điệp qua Linux kernel

Internet Protocol (IP) được coi là trái tim của hệ thống thông tin mạng trong Linux. Trong khi Linux được gắn chặt với khái niệm các tầng (như transport, network, ... điều này giúp cho nó có thể sử dụng một giao thức khác như ATM) thì IP luôn đi kèm với khái niệm về các gói tin. Hoạt động của IP trong tầng mạng bằng cách routing (dẫn đường) và forwarding (chuyển tiếp) cũng như encapsulating (bọc dữ liệu), hình trên đây giúp cho bạn hiểu được cách Linux kernel chuyển các gói tin qua.

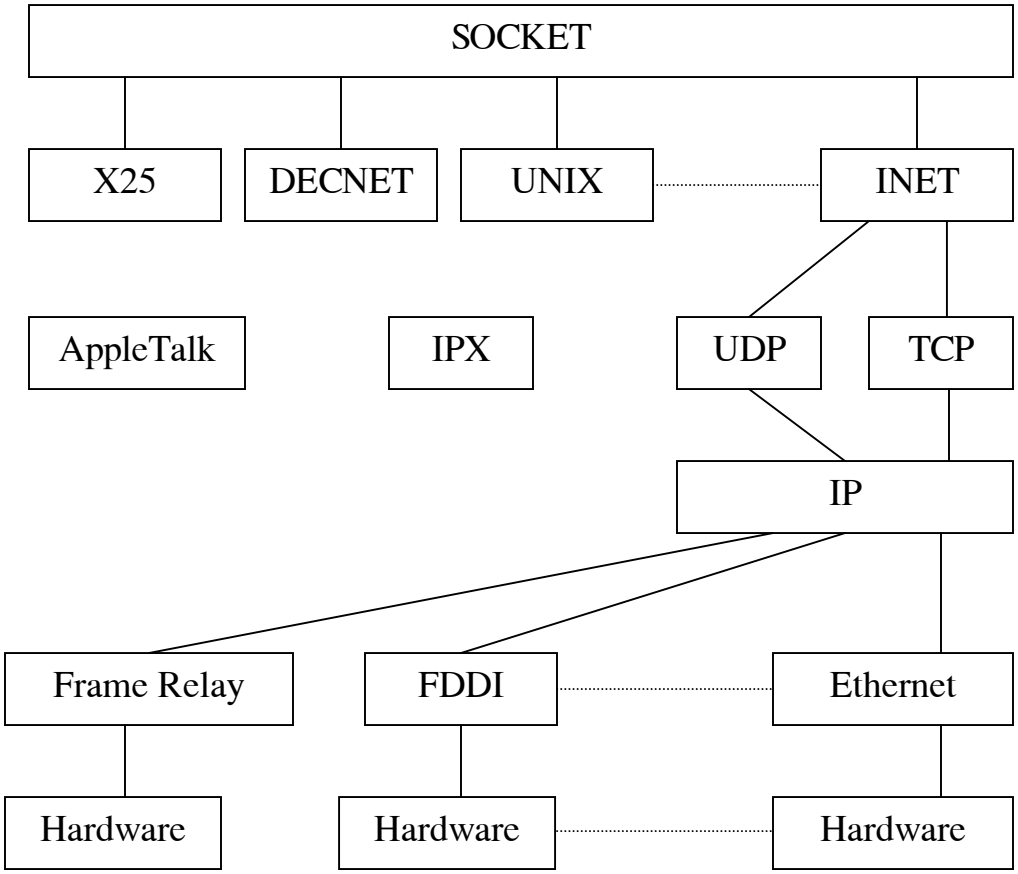
Khi một ứng dụng cần truyền thông, nó gửi các gói tin thông qua các socket tới tầng Giao vận (transport) (TCP hoặc UDP) và sau đó gói tin được gửi tới tầng mạng (Network layer-IP). Trong tầng mạng, nhân tiến hành tìm kiếm tuyến (đường dẫn) tới máy tính đích thông qua bảng định tuyến (routing cache) hoặc từ thông tin chuyển gói của nó (FIB - Forwarding Information Base). Nếu gói tin được chuyển cho một máy tính khác, kernel đánh địa chỉ cho gói và gửi nó tới giao diện truyền đi tầng liên kết (link layer output interface) (thường là thiết bị Ethernet), nơi cuối cùng thực hiện việc gửi gói tin ra thiết bị vật lý.

Khi gói tin được truyền qua thiết bị vật lý, giao diện tiếp nhận nhận được nó và kiểm tra xem gói tin này có thực sự gửi cho mình hay không. Nếu đúng, nó sẽ gửi gói tin này lên tầng IP, tầng này sẽ thực hiện tìm đường đi đến đích của gói. Nếu gói được chuyển tới một máy tính khác, tầng IP gửi nó trả lại cho giao diện truyền đi (output interface). Nếu gói tin được gửi cho chương trình ứng dụng, tầng IP chuyển nó lên tầng giao vận và socket của chương trình ứng dụng sẽ đọc gói khi nó sẵn sàng.

Theo cách này, mỗi socket và giao thức thực hiện các hàm định dạng và kiểm tra khác nhau. Tất cả quá trình được thực hiện cùng với các tham chiếu và bảng và các bảng điều khiển (jump table) để phân tách các thủ tục và phần lớn trong chúng được khởi tạo trong quá trình khởi động của máy tính. Mục sau sẽ trình bày chi tiết quá trình khởi tạo này.

1.2 Chồng giao thức (Protocol Stack)

Protocol Stack là một phần trong kernel code. Với cấu trúc giao thức mạng theo cách này làm cho nó có thể hỗ trợ rất nhiều giao thức trong cùng một thời điểm. Trong phần này chúng ta chỉ lưu ý chủ yếu đến TCP/IP protocol stack và nó sẽ được phân tích và mô phỏng ở phần dưới đây.



Hình 2: Cấu trúc protocol stack của Linux

Hình trên là cấu trúc Protocol Stack của Linux. Cấu trúc này làm cho nó có thể hỗ trợ nhiều giao thức mạng trong Linux kernel. Giao thức TCP/IP được thiết lập bởi 5 tầng:

- ***SOCKET layer:*** Là giao diện giữa các chương trình ứng dụng và các tầng thấp hơn, nó bao gồm tất cả các giao thức mạng khác nhau cho các chương trình ứng dụng. Các socket BSD là các cấu trúc trừu tượng hơn mà nó chứa các socket INET. Ứng dụng đọc từ hoặc ghi ra các sockets BSD. Các sockets BSD chuyển các thao tác thành các thao tác của socket INET. Các ứng dụng được chạy trong không gian người dùng, là mức trên cùng của chồng giao thức; chúng giống như một giao tiếp kết nối 2 chiều hoặc phức tạp như Giao thức Thông tin định tuyến (Routing Information Protocol - RIP).
- ***INET layer:*** IP Specific INET Socket là các thành phần dữ liệu và cài đặt cụ thể của các socket nói chung. Chúng được gắn với các hàng đợi và mã thi hành các thao tác của socket, chẳng hạn như đọc, ghi và tạo kết nối. Thực tế

chúng là thành phần trung gian giữa socket chung của ứng dụng và giao thức tầng Giao vận.

- ***TCP (UDP) layer:*** TCP và UDP là các giao thức thông dụng nhất trong tầng Giao vận. UDP đơn giản chỉ cung cấp một cơ cấu để chỉ các gói tới các cổng trong máy tính. Trong khi TCP cho phép các kết nối phức tạp hơn dựa trên các thao tác, bao gồm các kỹ thuật khôi phục các gói bị mất và quản lý truyền thông. Cả hai đều sao chép phần payload của gói giữa người dùng và nhân. Dù sao, cả hai giao thức này đều là một phần tầng trung gian giữa các ứng dụng và Mạng.
- ***IP layer:*** IP là một giao thức tầng Mạng chuẩn. Nó kiểm tra các gói tin đến, xem chúng được gửi cho chính máy tính của mình hay cần phải chuyển (forward) chúng. Nó phân nhỏ các gói tin nếu thấy cần thiết và phân phối chúng cho các giao thức tầng Giao vận. Chúng duy trì một cơ sở dữ liệu định tuyến cho các gói dữ liệu ra; đánh địa chỉ và chia nhỏ các gói (nếu cần) trước khi gửi chúng xuống tầng liên kết.
- ***Network device layer:*** Đây là tầng cuối trong protocol stack; chúng sử dụng thủ tục của tầng liên kết (thông thường là Ethernet) để truyền thông với các thiết bị khác trong việc chuyển hay nhận dữ liệu. Các giao diện tiếp nhận (input interface) copy các gói dữ liệu từ môi trường truyền dẫn (medium), thực hiện một vài phép kiểm tra, sau đó chuyển tiếp nó lên tầng mạng. Giao diện truyền đi nhận gói từ tầng mạng, thực hiện một vài phép kiểm tra và chuyển nó ra môi trường truyền dẫn.

Các ứng dụng ở tầng trên cùng có thể đơn giản là việc chating 2 chiều, hoặc phức tạp như RIP.

1.3 Cấu trúc của gói

Phương pháp để bảo vệ tầng các giao thức nghiêm ngặt mà không lãng phí thời gian copy các tham số và payload là tổ chức cấu trúc dữ liệu gói chung (một bộ đệm socket sk_buff). Thông qua tất cả các con trỏ hàm khác nhau, dữ liệu được truyền qua các giao thức, phần dữ liệu payload chỉ được copy 2 lần; một từ ứng dụng người dùng vào không gian nhân và một từ không gian nhân tới thiết bị ra (cho các gói ra ngoài).

Cấu trúc này chứa các con trỏ tới tất cả các thông tin về một gói - socket, thiết bị, tuyến, vị trí dữ liệu, vv... Các giao thức của tầng Giao vận tạo các cấu trúc gói này từ bộ đệm đầu ra, và ngược lại các trình điều khiển thiết bị tạo ra chúng từ dữ liệu đến. Mỗi một tầng sau đó sẽ điền thêm thông tin cần thiết của nó để phục vụ cho việc xử lý gói. Tất cả các giao thức - tầng Giao vận (TCP/UDP), tầng Internet (IP), Liên kết (Ethernet) - sử dụng cùng một socket buffer.

sk	con trỏ tới chính socket
stamp	thời gian đến
dev	con trỏ tới thiết bị nhận/truyền
h	con trỏ tới phần header của tầng Giao vận
nh	con trỏ tới phần header tầng Mạng
mac	con trỏ tới phần header tầng Liên kết
dst	con trỏ tới dst_entry
cb	thông tin điều khiển cho mỗi gói của TCP
len	độ dài dữ liệu thực
csum	checksum
protocol	giao thức mạng của gói
truesize	kích thức bộ đệm
head	con trỏ tới đầu của bộ đệm
data	con trỏ tới đầu dữ liệu
tail	con trỏ phần đuôi
end	con trỏ tới cuối
destructor	con trỏ tới hàm destruct

Hình 3: Cấu trúc gói (sk_buff)

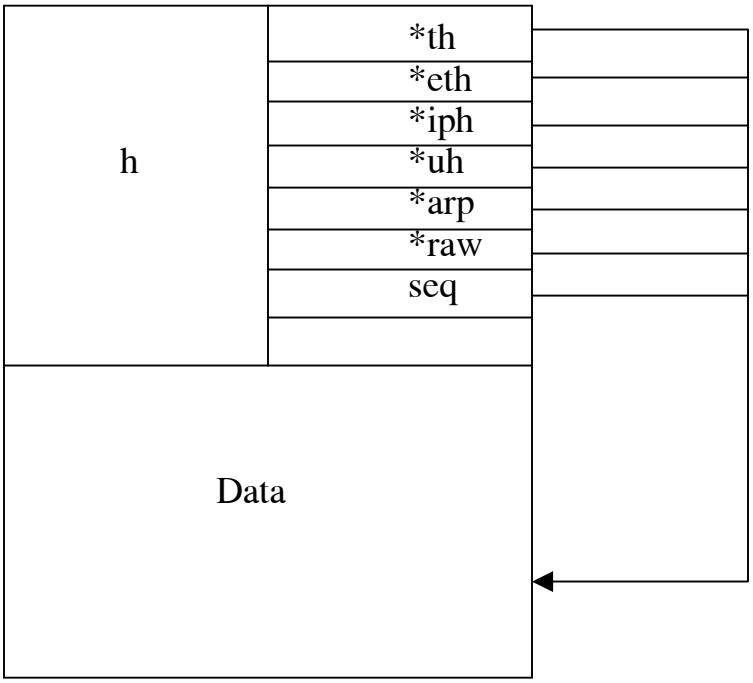
Để hiểu về hệ thống mạng trong Linux, thì một điều quan trọng nhất là việc sử dụng cấu trúc dữ liệu trong Linux thông qua cấu trúc sk_buff. Cấu trúc này được định nghĩa trong include/linux/skbuff.h (trong kernel source). Các sk_buff được sử dụng để quản lý truyền thông các gói. Mỗi một sk_buff là một cấu trúc điều khiển được chia cho một khối bộ nhớ. Cấu trúc điều khiển chứa các con trỏ tới thông tin header trong cấu trúc dữ liệu được gán.

Cấu trúc sk_buff rất lớn, nhưng một số thành phần cấu trúc dữ liệu (quan trọng) được mô tả chi tiết dưới đây:

```
struct sk_buff {
      /* pointers in doubly linked list (next and
      previous buffer in list)*/
      struct sk_buff    * next, * prev;

      /* List we are on */
      struct sk_buff_head * list;

      /* Socket which owns this sk_buff */
      struct sock *sk;

      /* Time we arrived */
      struct timeval    stamp;

      /* Device we arrived on/are leaving by */
      struct device     *dev;

      /* Transport layer header */
      union
      {
            struct tcphdr     *th;
            struct udphdr     *uh;
            struct icmphdr    *icmph;
            struct igmphdr    *igmph;
            struct iphdr      *ipiph;
            struct spxhdr     *spxh;
            unsigned char     *raw;
      } h;

      /* Network layer header */
      union
      {
            struct iphdr      *iph;
            struct ipv6hdr    *ipv6h;
            struct arphdr     *arph;
            struct ipxhdr     *ipxh;
            unsigned char     *raw;
      } nh;

      /* Link layer header */
      union
      {
            struct ethhdr     *ethernet;
            unsigned char     *raw;
      } mac;
......
};
```

Linux sử dụng liên kết giữa 2 danh sách sk_buff (next và prev) như là tuyến các buffer (linear buffers). Điều này có sự khác nhau giữa BSD và Unix (sử dụng một chuỗi các buffer nhỏ - mbufs). Linear buffers sử dụng tốn bộ nhớ hơn mbufs,

nhưng lại sử lý rất nhanh nên nó được sử dụng trên network buffer (ví dụ như nối thêm vào sau hoặc xoá bỏ khi khởi động).



Hình 4: sk_buff chứa các con trỏ tới thông tin header.

Nhìn vào định nghĩa cấu trúc dữ liệu ở trên thì sk_buff bao gồm các con trỏ tới thông tin header tại các tầng liên kết (link layer), tầng mạng (network layer), và tầng giao vận (transport layer), do vậy đây là vấn đề chính giải quyết việc quản lý bộ nhớ của các gói trong Linux kernel.

Linux kernel bao gồm các hàm tiện ích phục vụ cho việc quản lý danh sách các sk_buff. Điều này thuận lợi khi giải quyết việc 2 tuyến cùng hoạt động đồng thời trên cùng một khối bộ nhớ. Những hoạt động này có thể mô tả như sau:

- skb_dequeue(), lấy buffer đầu tiên trong danh sách.
- skb_queue_head(), đặt một buffer vào đầu một danh sách.
- skb_queue_tail(), đặt một buffer vào cuối của một danh sách.
- skb_unlink(), xoá bỏ một buffer trong danh sách chứa nó (buffer không rỗng, phải được xoá trong danh sách).
- skb_insert(), đặt một buffer trước một buffer đã chỉ ra trong danh sách (thường sử dụng cho các giao thức như TCP, phục vụ cho việc thay đổi lại thứ tự các buffer cho đúng thứ tự gói).
- skb_append(), đặt một buffer vào sau một buffer đã chỉ ra trong danh sách (TCP).
- alloc_skb(), tạo một sk_buff mới và thiết lập các giá trị ban đầu cho nó.

- kfree_skb(), giải phóng khỏi buffer (giải phóng bộ nhớ).

Dưới đây đưa ra một đoạn mã đơn giản để quản lý một danh sách các buffers (tham khảo trong tài liệu Network Buffers and Memory Management).

```
void append_frame(char *buf, int len)
   {
         /* Cấp phát một sk_buff */
         struct sk_buff *skb = alloc_skb(len, GFP_ATOMIC);

         if (skb == NULL) {
               /* Không thể cấp phát sk_buff, gói tin bị huỷ */
               my_dropped++;
         }
         else {
               /* Dành len bytes dữ liệu trong sk_buffs */
               skb_put(skb, len);

               /* Copy bộ đệm vào sk_buff*/
               memcpy(skb->data, dat, len);

               /* Kết nối danh sách sk_buff */
               skb_append(&my_list, skb);
         }
   }
void process_queue(void)
{
   struct sk_buff    *skb;
   /* Đưa ra sk_buff đầu tiên trong danh sách kết nối */
   while((skb = skb_dequeue(&my_list)) != NULL) {
         process_data(skb);
         /*  Giải phóng sk_buff sau khi đã sử lý data */
         kfree_skb(skb, FREE_READ);
   }
}
```

Giải thích: ở trên đưa ra cách sử dụng hàm quản lý bộ nhớ của sk_buff, với hàm append_frame() là một đoạn mã trong trình điều khiển thiết bị mạng, tương ứng với hàm netif_rx() có trong tệp net/core/dev.c, hàm này nhận một gói tin từ trình điều khiển thiết bị và đưa ra hàng đợi cho mức giao thức cao hơn. Hàm process_queue() tương ứng với hàm net_bh() có trong tệp net/core/dev.c, đưa sk_buff ra hàng đợi để xử lý ở mức cao hơn.

1.4 Định tuyến Internet (Internet Routing)

Tầng IP điều khiển định tuyến giữa các máy tính. Nó giữ 2 cấu trúc dữ liệu; một là Bảng thông tin chuyển gói (Forwarding Information Base - FIB) chứa tập hợp các thông tin chi tiết về các tuyến đã biết, và một bộ đệm định tuyến nhanh hơn cho các địa chỉ đích hiện đang sử dụng. (Cũng có một cấu trúc thứ ba - neighbor table - giữ tập hợp các máy tính được kết nối vật lý tới máy).

Bảng FIB là bảng tham chiếu định tuyến gốc; nó chứa tới 32 vùng (mỗi vùng ứng với một bit trong địa chỉ IP) và các entry cho tất cả các địa chỉ đích đã biết. Mỗi vùng chứa các entry cho mạng và máy mà có thể được định danh duy nhất bởi số bit xác định - một mạng với netmask 255.0.0.0 có 8 bit có nghĩa nên nằm trong vùng 8, hoặc một mạng với netmask 255.255.255.0 có 24 có nghĩa nên nằm trong vùng 24. Khi IP cần tìm một đường, nó bắt đầu với các vùng đặc biệt nhất và tìm kiếm toàn bộ bảng cho đến khi tìm thấy (có ít nhất một entry ngầm định). File **/proc/net/route** chứa nội dung của FIB.

Bộ đệm định tuyến (routing cache) là một bảng hash mà IP sử dụng route các gói trong thực tế. Nó có thể chứa tới 256 chuỗi entry định tuyến hiện hành, với mỗi vị trí của entry được quyết định bởi một hàm hash. Khi một máy cần gửi một gói, IP tìm một entry trong bộ đệm định tuyến. Nếu không có, nó tìm một tuyến phù hợp trong FIB và chèn entry mới này vào bộ đệm. (Entry này để giao thức sử dụng việc định tuyến, không phải entry của FIB). Các entry còn lại trong bộ đệm đến chừng nào chúng còn được sử dụng; nếu không có truyền thông tới địa chỉ đích, entry không có hiệu lực và IP xóa nó khỏi bộ đệm. File **/proc/net/rt_cache** lưu trữ nội dung của bộ đệm định tuyến này.

Các bảng này cho phép tất cả các tuyến trên hệ thống thông thường. Thậm chí các giao thức khác (chẳng hạn RIP) sử dụng cùng cấu trúc; chúng chỉ sửa các bảng tồn tại trong nhân sử dụng hàm **ioctl()** (được đề cập ở phần sau).

2. Khởi tạo mạng

Phần này trình bày tổng quát về quá trình khỏi tạo mạng khi hệ điều hành mạng khởi động, sử dụng chương trình **ifconfig** và **route** để thiết lập liên kết mạng. Cuối cùng, để tiện cho những độc giả quan tâm đến việc lập trình, chúng tôi đưa ra các thủ tục hàm liên quan đến các chương trình ifconfig và route.

2.1 Tổng quan

Linux sẽ chỉ khởi tạo bảng định tuyến khi khởi động nếu máy tính đã được cấu hình mạng (hầu hết các máy Linux làm việc với mạng, thậm chí máy stand-alone, nếu chỉ sử dụng loopback). Khi nhân đã được nạp xong, nó chạy một tập các chương trình tiện ích đặc biệt - mà thực hiện đọc các file cấu hình, thiết lập các tính năng mạng của máy tính. Các công việc này bao gồm việc quyết định địa chỉ của máy, khởi tạo các giao diện của nó (chẳng hạn các card Ethernet), và thêm các định tuyến tĩnh đã biết hoặc quan trọng (ví dụ định tuyến tới một router được kết nối với Internet). Nếu bản thân máy tính là một router, nó có thể thi hành một chương trình mà cho phép cập nhật bảng định tuyến của nó.

Toàn bộ quá trình cấu hình có thể là tĩnh hoặc động. Nếu các địa chỉ và tên không bao giờ (hoặc không thường xuyên) thay đổi, người quản trị hệ thống phải định nghĩa các tùy chọn và các biến trong các file khi thiết lập hệ thống. Trong môi trường hay biến đổi hơn, có thể sử dụng một giao thức DHCP (Dynamic

Hardware Configuration Protocol) để hỏi các thông tin về địa chỉ, router, và DNS Server để cấu hình khi nó khởi động.

2.2 Khởi động

Khi Linux khởi động như một hệ điều hành, nó nạp ảnh của nó từ đĩa vào trong bộ nhớ, giải nén, và tự thiết lập bằng cách cài đặt hệ thống file và quản lý bộ nhớ và các hệ thống khác. Nhiệm vụ sau cùng của nhân khi khởi động là nó thực hiện trình ***init***. Trình này đọc một file cấu hình (**/etc/inittab**) và thi hành các script khởi động (được tìm trong **/etc/rc.d** trong các bản phân phối của RedHat). Có thể có rất nhiều các file script được thi hành, trong đó có file kịch bản khởi động mạng (**/etc/rc.d/init.d/network**).

Kich bản khởi đông mang

Kịch bản khởi động mạng thiết lập các biến môi trường để định danh tên máy tính và thiết lập hoặc không thiết lập mạng cho máy tính. Phụ thuộc vào các giá trị được đưa ra, script network bật (hoặc tắt) **IP forwarding** và **IP fragmentation**. Nó cũng thiết lập định tuyến ngầm định cho tất cả các giao dịch mạng và thiết bị sử dụng để gửi giao dịch đó. Cuối cùng, nó đánh thức các thiết bị mạng sử dụng các chương trình *ifconfig* và *route*. (Trong môi trường động, nó có thể truy vấn DHCP server để lấy thông tin mạng thay vì đọc các file của nó).

Các kịch bản được thi hành khi thiết lập mạng có thể không phức tạp lắm; hoàn toàn có thể chuyển thành một file script lớn, thi hành dãy các lệnh để thiết lập đúng cho một máy tính đơn lẻ. Tuy nhiên, hầu hết các nhà phân phối Linux đều đưa ra một số lớn các kịch bản mang tính chất chung, làm việc với nhiều dạng máy. Điều này để lại một số hành động gián tiếp và việc thi hành có điều kiện, song thực sự làm cho việc cài đặt được dễ hơn. Ví dụ: trong bản phân phối Red Hat, kịch bản **/etc/rc.d/init.d/network** chạy một vài kịch bản khác và thiết lập các biến như **interfaces_boot** để lưu dấu vết xem đã chạy kịch bản **/etc/sysconfig/network-scripts/ifup** nào. Việc theo dõi tiến trình này một cách thủ công cũng khá phức tạp, nhưng một số sửa đổi đơn giản trên 2 file cấu hình (đưa đúng tên và địa chỉ IP vào trong file **/etc/sysconfig/network** và **/etc/sysconfig/network-script/ifcfg-eth0**) sẽ thiết đặt cả hệ thống hoạt động đúng.

Khi kịch bản khởi động mạng kết thúc, FIB chứa các định tuyến đặc biệt cho các máy hoặc các mạng đã biết, còn bộ đệm định tuyến và bảng neighbor đều rỗng. Khi truyền thông bắt đầu truyền, nhân sẽ cập nhật bảng neighbor và bộ đệm định tuyến như một phần của quá trình hoạt động mạng thông thường.

ifconfig (interface configuration)

Chương trình này phục vụ cho việc cấu hình các giao diện thiết bị mạng để có thể sử dụng. Đây là một chương trình được sử dụng rất nhiều, và nó không phải là một phần của nhân. Nó cũng cấp mỗi thiết bị các thông tin địa chỉ, netmask, và địa chỉ phát tán (broadcast address). Thiết bị lần lượt chạy các hàm khởi tạo của nó

(để thiết lập các biến tĩnh) và đăng ký các ngắt của nó và các thủ tục dịch vụ với nhân. Lệnh **ifconfig** trong kịch bản mạng có dạng như sau:

ifconfig ${DEVICE} ${IPADDR} netmask ${NMASK} broadcast ${BCAST}
(mà các biến hoặc được ghi trực tiếp trong kịch bản hoặc được định nghĩa trong các kịch bản khác).

Chương trình **ifconfig** cũng có thể cung cấp thông tin về các thiết bị mạng đã cấu hình hiện hành (khi gọi chương trình không có tham số, nó sẽ hiển thị tất các giao diện được kích hoạt hiện hành; gọi với tùy chọn -a nó hiển thị tất cả các giao diện, cả kích hoạt hoặc không kích hoạt).

Chương trình này cung cấp tất cả các thông tin khả dụng về mỗi giao diện mạng; địa chỉ, trạng thái, thống kê gói, và các đặc trưng của hệ điều hành. Thông thường sẽ có ít nhất hai giao diện - một card mạng và thiết bị loopback. Thông tin cho mỗi giao diện có dạng như sau:

```
$ifconfig -a
eth0      Link encap:Ethernet  HWaddr 00:80:AD:8D:C7:4A
inet addr:200.1.1.6  Bcast:200.1.1.255  Mask:255.255.255.0
UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
RX packets:344 errors:0 dropped:0 overruns:0 frame:0
TX packets:4 errors:6 dropped:0 overruns:0 carrier:12
collisions:102 txqueuelen:100
Interrupt:11 Base address:0x2040
```

Người dùng có đặc quyền có thể sử dụng lệnh **ifconfig** để thay đổi các thiết lập của giao diện từ dòng lệnh, với cú pháp như sau:

ifconfig interface [aftype] options | address ...

Ví dụ:
ifconfig eth0 down : shutdown eth0
ifconfig eth1 up : kích hoạt giao diện eth1
ifconfig eth0 arp : bật ARP trên giao diện eth0
ifconfig eth0 -arp : đóng ARP trên giao diện eth1
ifconfig eth0 netmask 255.255.255.0: thiết lập netmask cho eth0
ifconfig lo mtu 2000 : thiết lập đơn vị truyền lớn nhất cho giao diện loopback.
ifconfig eth1 172.16.0.7 : thiết lập địa chỉ IP cho eth1.

Chú ý rằng thay đổi cấu hình giao diện mạng có thể thay đổi bảng định tuyến một cách gián tiếp. Ví dụ, thay đổi netmask có thể thay đổi một số tuyến.

route

Chương trình route đơn giản thêm định tuyến xác định trước cho các thiết bị giao diện vào bảng thông tin chuyển FIB (Forwarding Information Base). Đây là một chương trình người dùng, mà các lệnh được sử dụng trong kịch bản mạng có dạng:

route add -net ${{NETWORK} netmask ${NMASK} dev ${DEVICE}

hoặc

route add -host ${IPADD} ${DEVICE}

(các biến được chỉ ra hoặc định nghĩa trước trong các kịch bản khác)

Chương trình route cũng có thể xóa định tuyến (nếu chạy với tùy chọn del) hoặc cung cấp thông tin về định tuyến mà hiện hành đã được định nghĩa (khi gọi chương trình không có tùy chọn route). Nó sẽ hiển thị bảng định tuyến IP của nhân (trong FIB, không xóa bộ đệm định tuyến). Ví dụ:

```
$route -n
Kernel IP routing table
Destination Gateway   Genmask       Flags MetricRef   Use   Iface
172.16.1.4  *         255.255.255.255 UH   0      0      0     eth0
172.16.1.0  *         255.255.255.0   U    0      0      0     eth0
127.0.0.0   *         255.0.0.0       U    0      0      0     lo
default   viper.u.edu0.0.0.0          UG   0      0      0     eth0
```

Siêu người dùng có thể sử dụng route để thêm và xóa các định tuyến IP từ dòng lệnh với cú pháp như sau:

route add [-net | -host] target [option arg]

route del [-net | -host] target [option arg]

Ví dụ:

route add -host 127.16.1.0 eth1 : thêm định tuyến cho máy.

route add -net 172.16.1.0 netmask 255.255.255.0 eth0 : thêm một mạng

route add default gw jeep : thiết lập tuyến ngầm định đến jeep

route del -host 172.16.1.16 : xóa entry cho máy 172.16.1.16

Các chương trình định tuyến động

Nếu máy tính của bạn là một router, kịch bản mạng sẽ chạy một chương trình dạng **routed** hoặc **gated**. Trong hầu hết các máy tính không chạy một trong các chương trình này.

2.3 Các hàm liên quan

Dưới đây chúng tôi sẽ đưa ra danh sách các hàm của Linux kernel và chương trình mạng quan trọng (trong gói **net-tools**) liên quan trực tiếp đến việc khởi tạo mạng, đồng thời nêu rõ chức năng của chúng.

Ifconfig

Funtions/File	Descriptions
devinet_ioctl()	-tạo ra cấu trúc thông tin yêu cầu (ifreq) và copy dữ liệu từ

có trong tệp net/ipv4/devinet.c	không gian người dùng tới không gian kernel -nếu nó là yêu cầu hay hành động ở mức INET thì thực thi nó -nếu nó là yêu cầu hay hành động của thiết bị thì gọi một hàm hàm để copy ifreq trở lại bộ nhớ người dùng -hàm này trả về 0 khi thành công.
ifconfig main() có trong SOURCES/ifconfig.c	-mở một socket (chỉ sử dụng với hàm ioctl) -tìm các tham số dòng lệnh -gọi if_print() nếu không có tham số hoặc chỉ có tham số chỉ tên giao diện -tìm theo các tham số còn lại, thiết lập hay bỏ cờ hoặc nó gọi hàm ioctl() để thiết lập các biến cho giao diện.
if_fetch() trong tệp SOURCES/lib/interface.c	-điền một cấu trúc giao diện với việc thực hiện gọi các hàm ioctl() cho các cờ, địa chỉ phần cứng, metric, MTU, map, và các thông tin về địa chỉ.
if_print() trong tệp SOURCES/ifconfig.c	-gọi hàm ife_print() cho một (hay tất cả các) giao diện cho trước để lấy các giao diện, gọi hàm if_readlist() để điền danh của cấu trúc nếu cần thiết và sau đó hiển thị thông tin về từng giao diện.
if_readlist() trong tệp SOURCES/lib/interface.c	-mở tệp /pro/net/dev và phân tích dữ liệu trong các cấu trúc giao diện - gọi hàm add_interface() cho từng thiết bị để đưa các cấu trúc vào trong danh sách.
inet_ioctl() trong tệp net/ipv4/af_inet.c	thực hiện việc nhảy dựa vào lệnh đã phân tích (gọi hàm devinet_ioctl()).
ioctl()	nhảy tới hàm điều khiển inet_ioctl().

route

-INET_rinput()trong tệp SOURCES/lib/inet_sr.c	-thực hiện việc kiểm tra các lỗi (không xóa hay chữa bảng routing cache) -và gọi hàm INET_setroute().
INET_rprint() trong tệp SOURCES/lib/inet_gr.c	-nếu cờ FIB được thiết lập, gọi hàm rprint_fib() (đọc, phân tích và hiển thị nội dung tệp /proc/net/route) -nếu cờ CACHE được thiết lập, thì gọi hàm rprint_cache() (đọc , phân tích và hiển thị nội dung của tệp /proc/net/rt_cache).
INET_setroute() có trong tệp SOURCE/lib/inet_sr.c	-phục vụ việc thiết lập tuyến tới một mạng hoặc là một máy -kiểm tra xem các địa chỉ có hợp lý không? -duyệt qua các tham số, điền vào cấu trúc rtentry -kiểm tra sự xung đột của mặt nạ mạng (netmask) -tạo một socket tạm thời -gọi hàm ioctl() với cấu trúc rtentry trên để thêm hoặc là xoá tuyến đã định -đóng socket và trả về giá trị 0.
ioctl()	thực hiện gọi hàm ip_rt_ioctl().

`ip_rt_ioctl()` trong tệp `net/ipv4/fib_frontend.c`	-chuyển đổi các tham số đã qua tới routing table entry (cấu trúc `rtentry`) -nếu xoá một tuyến, thì gọi hàm `fib_get_table()` để tìm bảng tương ứng và gọi hàm `tb_delete()` để xoá nó - nếu thêm vào một tuyến, thì gọi hàm `fib_net_table()` để tìm kiếm một entry, gọi hàm `>tb_insert()` để thêm vào entry, trả về 0 nếu thành công.
`route main()` trong tệp `SOURCES/route.c`	-gọi các thủ tục khởi động để thiết đặt các chức năng soạn thảo và in -lấy và phân tích tuỳ chọn trong dòng lệnh -kiểm tra các tuỳ chọn -nếu không có tuỳ chọn nào thì gọi hàm `route_info(), -` -nếu có tuỳ chọn add, del hay flush tuyến, thì gọi hàm `route_edit()` với các tham số đã được truyền -nếu các tuỳ chọn đưa ra sai, thì hiển thị help -trả về giá trị 0 khi thành công.
`route_edit()` trong tệp `SOURCES/lib/setroute.c`	-gọi hàm `get_aftype()` để chuyển đổi địa chỉ từ dạng text về một con trỏ -kiểm tra các lỗi (không hỗ trợ hoặc họ không tồn tại) -gọi hàm `rinput()` với địa chỉ đó qua hàm `INET_rinput()`.
`route_info()` trong tệp `SOURCES/lib/getroute.c`	-thực hiện gọi hàm `get_aftype()` để dịch địa chỉ từ dạng text về một con trỏ -kiểm tra các lỗi -gọi hàm `rinput()` với địa chỉ đó qua hàm `INET_rinput()`.

3. Kết nối (Connection)

Phần này trình bày về quá trình kết nối; tổng quan về quá trình kết nối, mô tả về cấu trúc dữ liệu socket, giới thiệu về hệ thống định tuyến, và tổng quát hóa mã bổ sung trong nhân.

3.1 Tổng quan

Dạng mạng đơn giản nhất là kết nối giữa 2 máy tính với nhau. Trên mỗi đầu, một ứng dụng tạo một socket, tạo kết nối tầng giao vận, và sau đó gửi hoặc nhận các gói. Trong Linux, một socket thực bao gồm 2 cấu trúc socket (socket này chứa một socket khác). Khi một ứng dụng tạo một socket, nó đã được khởi tạo nhưng rỗng. Khi socket tạo kết nối, tầng IP lựa chọn đường tới máy cần truyền và lưu giữ thông tin đó vào socket. Khi này, tất cả truyền thông sử dụng kết nối đó với đường (tuyến) đó - các gói gửi sẽ chuyển qua thiết bị và định tuyến tới đúng máy ở xa, và các gói nhận được sẽ xuất hiện trong hàng đợi của socket.

3.2 Cấu trúc của Socket

Có 2 cấu trúc socket chính trong Linux: các socket BSD chung và các socket INET riêng của IP. Chúng có mối quan hệ chặt chẽ với nhau; mỗi một

socket BSD có một socket INET như là một phần dữ liệu của nó và ngược lại mỗi một socket INET có một socket BSD như là chủ của nó.

Các socket BSD có kiểu **struct socket** được định nghĩa trong **include/linux/socket.h**. Các biến socket BSD thường có tên là sock hoặc bắt đầu bằng từ này. Cấu trúc này chỉ có một số ít entry, các entry quan trọng nhất được mô tả ở dưới.

- **struct proto_ops *ops**: cấu trúc này chứa các con trỏ tới các hàm đặc trưng của giao thức cho cách đối xử của socket. Ví dụ: ops->sendmsg trỏ tới hàm inet_sendmsg().
- **struct inode *inode**: cấu trúc này trỏ tới file inode được gắn với socket này.
- **struct sock *sk**: đây là socket của INET, được gắn với socket này.

Các socket INET có kiểu struct sock như đã định nghĩa trong **include/net/sock.h**. Các biến socket INET thường có tên là sk hoặc bắt đầu bằng từ đó. Cấu trúc này có nhiều entry liên quan tới cách sử dụng khác nhau; có nhiều trường phụ thuộc cấu hình. Các thành phần dữ liệu quan trọng nhất được mô tả ở dưới:

- **struct sock *next, *pprev**: tất cả các socket được liên kết bởi các giao thức khác nhau, vì vậy các con trỏ này cho phép các giao thức đi ngang qua chúng.
- **struct dst_entry *dst_cache**: đây là con trỏ tới định tuyến bên kia của socket (địa chỉ đích của gói gửi).
- **struct sk_buff_head receive_queue**: đây là phần đầu của hàng đợi nhận.
- **struct sk_buff_head write_queue**: đây là phần đầu của hàng đợi gửi.
- **__u32 saddr**: địa chỉ nguồn (Internet) cho socket này.
- **struct sk_buff_head back_log, error_queue**: hàng đợi mở rộng cho các gói bị ùn lại (không lộn xộn với hàng đợi backlog) và các gói bị truyền sai cho socket này.
- **struct proto *prot**: cấu trúc này chứa các con trỏ tới các hàm đặc trưng cho giao thức tầng giao vận. Ví dụ: prot->recvmsg có thể trỏ tới hàm tcp_v4_recvmsg().
- **union struct tcp_op af_tcp; tp_pinfo**: các tùy chọn TCP cho socket này.
- **struct socket *sock**: socket cha BSD.
- Chú ý rằng có rất nhiều trường trong cấu trúc này; các trường này không quan trọng lắm hoặc có bản thân tên mang tính giải thích (ví dụ: ip_ttl là bộ đếm IP Time-To-Live).

3.3 Socket và định tuyến

Các socket chỉ đi qua **tiến trình tìm đường đi** cho mỗi địa chỉ đích (tại thời điểm kết nối). Bởi vì các socket của Linux có quan hệ chặt chẽ với IP, chúng chứa các tuyến tới phần kết nối khác (trong biến **sock->sk->dst_cache**). Các giao thức của tầng Giao vận gọi hàm **ip_route_connect()** để xác định đường đi từ máy

nguồn tới máy đích trong quá trình kết nối; sau đó định tuyến này được coi là không thay đổi (dù đường dẫn được trỏ bởi dst_cache có thể thay đổi thực sự). Socket không cần thực hiện tiếp tìm kiếm bảng định tuyến cho mỗi gói nó gửi hoặc nhận; nó chỉ cố thử lại nếu có điều gì đó xảy ra (chẳng hạn máy tính đối tác đã bị down). Đây là lợi điểm của việc sử dụng các kết nối.

3.4 Quá trình kết nối

Thiết lập kết nối

Các chương trình ứng dụng thiết lập các socket với dãy các lệnh gọi hệ thống mà tìm kiếm các địa chỉ ở xa, thiết lập socket, và sau đó kết nối tới máy ở đầu kia.

```
/* look up host */
server=gethostbyname(SERVER_NAME);
/* get socket */
sockfd=socket(AF_INET, SOCK_STREAM, 0);
/* Set up addresss */
address.sin_family = AF_INET;
address.sin_port = htons(PORT_NUM);
memcpy(&address.sin_addr, server->h_addr,server->h_length);
/* connect to server */
connect(sockfd, &address, sizeof(address);
```

Hàm **gethostbyname()** tìm kiếm một máy (chẳng hạn viper.cs.u.edu) và trả về một cấu trúc chứa một địa chỉ Internet (IP). Điều này phục vụ cho việc định tuyến (vì máy có thể phải truy vấn mạng để tìm kiếm địa chỉ) và chuyển địa chỉ từ dạng text sang dạng tương thích với máy tính (số unsigned integer 4 bytes).

Lệnh **socket()** có nhiều thú vị hơn: tạo ra một đối tượng socket, với kiểu dữ liệu phù hợp (một sock cho socket INET) và khởi tạo nó. Socket chứa thông tin inode và các con trỏ đặc trưng cho giao thức trỏ tới các hàm mạng khác nhau. Nó cũng thiết lập hàng đợi ngầm định (đến, ra, lỗi, và backlog), dạng thông tin header cho socket TCP, và các thông tin khác.

Cuối cùng, lệnh **connect()** nhảy tới giao thức kết nối thông thường (ví dụ tcp_v4_connect() hoặc udp_connect()). UDP đơn giản thiết lập một tuyến tới đích (khi không có kết nối ảo). TCP thiết lập tuyến và sau đó bắt đầu tiến trình kết nối TCP, gửi một gói với kết nối phù hợp và thiết lập các cờ cửa sổ.

Nhiệm vụ của hàm Socket()

- Kiểm tra lỗi trong hàm gọi.
- Tạo (cấp phát bộ nhớ) đối tượng socket.
- Đưa socket vào danh sách INODE.
- Thiết lập các con trỏ tới các hàm giao thức (INET).

- Ghi các giá trị cho kiểu socket và họ giao thức.
- Thiết lập trạng thái socket để đóng.
- Khởi tạo hàng đợi gói.

Nhiệm vụ của hàm Connect()

- Kiểm tra lỗi.
- Lựa chọn định tuyến tới địa chỉ đích:
  - Kiểm tra bảng định tuyến xem có entry tồn tại chưa (trả lại1 entry nếu tồn tại).
  - Tìm địa chỉ trong FIB.
  - Tạo một entry mới cho bảng định tuyến.
  - Đưa entry mới vào bảng định tuyến và trở lại.
- Ghi con trỏ tới entry định tuyến trong socket.
- Gọi hàm kết nối đặc trưng cho giao thức (ví dụ: gửi gói kết nối TCP).
- Thiết lập trạng thái socket để truyền.

Đóng kết nối

Đóng socket khá đơn giản. Một ứng dụng gọi hàm close() trên một socket, nó sẽ gọi hàm sock_close(). Hàm này sẽ thay đổi trạng thái socket để hủy kết nối và gọi hàm giải phóng của INET socket. INET socket lần lượt xóa hàng đợi của nó và gọi hàm đóng của giao thức của tầng Giao vận, tcp_v4_close() hoặc udp_close(). Hàm này thực hiện bất kỳ các thao tác cần thiết nào và xóa tất cả các cấu trúc dữ liệu còn lại. Chú ý rằng nó không thay đổi định tuyến; socket hiện tại rỗng, song nó vẫn được tham chiếu tới địa chỉ đích và entry trong bộ đệm định tuyến cho đến khi nó được giải phóng.

Nhiệm vụ của hàm close()

- Kiểm tra lỗi (xem socket có tồn tại không?)
- Thay đổi trạng thái của socket để hủy kết nối.
- Thực hiện bất kỳ thao tác đóng giao thức nào (ví dụ: gửi một gói TCP với bit FIN được thiết lập).
- Giải phóng bộ nhớ cho cấu trúc dữ liệu (TCP/UDP và INET).
- Xóa socket từ danh sách INODE.

3.5 Các hàm của Linux

Dưới đây chúng tôi đưa ra danh sách (theo alphabet) các hàm quan trọng trong Linux kernel mà liên quan đến việc kết nối (bắt đầu tiến hành kết nối với lời gọi hàm **sock_creat(),** và đóng socket với lời gọi hàm **sock_close()**), đồng thời phân tích mã nguồn giúp cho việc tra cứu và modify.

Functions/tệp	Description
`destroy_sock()` có trong tệp	-thực hiện việc xoá một số bộ đếm thời gian

net/ipv4/af_inet.c	-gọi các hàm huỷ bỏ giao thức -giải phóng hàng đợi -tự giải phóng cấu trúc socket.
fib_lookup() có trong tệp include/net/ip_fib.h	-gọi hàm tb_lookup() [= fn_hash_lookup()] trên các bảng cục bộ và chính -trả về tuyến hoặc là lỗi không tìm thấy tuyến.
fn_hash_lookup() có trong tệp net/ipv4/fib_hash.c	-tìm kiếm và trả về tuyến là một địa chỉ.
inet_create() có trong tệp net/ipv4/af_inet.c	-gọi hàm sk_alloc() để lấy bộ nhớ cho sock -khởi tạo cấu trúc sock: +thiết lập cấu trúc proto với giá trị tương ứng cho TCP hoặc là UDP +gọi hàm sock_init_data() +thiết lập các biến family, protocol, ... -gọi hàm khởi tạo giao thức.
inet_release() có trong tệp net/ipv4/af_inet.c	-thay đổi trạng thái socket trở về không kết nối -gọi hàm ip_mc_drop_socket rời nhóm multicast (nếu cần) -thiết lập thành viên sở hữu dữ liệu của socket về NULL -gọi sk->prot->close()[=TCP/UDP_close()].
ip_route_connect()có trong tệp include/net/route.h	-gọi hàm ip_route_output() để lấy một địa chỉ đích trong lời gọi -trả về (nếu lời gọi được thực hiện hoặc sinh ra lỗi) -ngược lại, xoá con trỏ định tuyến và tiếp tục thực hiện lại.
ip_route_output() có trong tệp net/ipv4/route.c	-thực hiện việc tính giá trị hash cho địa chỉ -chạy từ hash qua table để tìm các địa chỉ và TOS (Type of Service) -nếu có sự trùng lặp, thì tiến hành cập nhật stats và trả về route entry -ngược lại thì gọi hàm ip_route_output_slow().
ip_route_output_slow() có trong tệp net/ipv4/route.c	-nếu địa chỉ nguồn đã biết thì tìm kiếm thiết bị ra chuẩn -nếu địa chỉ đích không biết thì thiết lập lookback -gọi hàm fib_lookup() để tìm tuyến trong FIB -cấp phát bộ nhớ cho bảng routing table entry mới -khởi tạo table entry với source, destination, TOS, thiết bị ra chuẩn, cờ -tiếp đến là gọi hàm rt_set_nexthop() để tìm đích tiếp theo -trả về hàm rt_intern_hash() hàm này tiến hành cài đặt tuyến trong bảng định tuyến.
rt_intern_hash() có trong tệp net/ipv4/route.c	-tiến hành vòng lặp từ giá trị hash tìm kiếm trong rt_hash_table -nếu tìm được keys, thì đưa rtable vào trước bucket -nếu không tìm thấy thì đưa rtable vào trong bảng hash table tại vị trí của giá trị hash.
sock close() có trong tệp	-thực hiện kiểm tra, nếu socket đã tồn tại (có thể NULL)

net/socket.c	-gọi hàm sock_fasync() để xoá socket từ trong danh sách async -gọi hàm sock_release().
sock_create() có trong tệp net/socket.c	-kiểm tra các tham số -gọi hàm sock_alloc() để nhận được inode đúng cho socket và khởi tạo nó -thiết lập socket->type (với các giá trị SOCK_STREAM, SOCK_DGRAM...) -gọi net_family->create() [= inet_create()] để xây dựng cấu trúc sock -trả về socket đã thiết lập.
sock_init_data() có trong tệp net/core/sock.c	-khởi tạo các giá trị ban đầu cho sock.
sock_release()có trong tệp net/socket.c	-thay đổi trạng thái về không kết nối -gọi sock->ops->release() [= inet_release()] -gọi hàm iput() để xoá socket trong danh sách inode.
sys_socket() có trong tệp net/socket.c	-gọi hàm sock_create() để tạo và thiết lập ban đầu socket -gọi hàm get_fd() để gán fd cho socket -thiết lập socket->file [=fcheck()] (để trỏ tới file) -cuối cùng là gọi hàm sock_release() nếu có lỗi.
tcp_close()có trong tệp net/ipv4/tcp.c	-thực hiện viêc kiểm tra lỗi -đưa ra bộ đệm và bỏ tất cả các gói từ hàng đến -gửi thông báo tới đích để đóng kết nối (nếu yêu cầu).
tcp_connect() có trong tệp net/ipv4/tcp_output.c	-hoàn tất gói kết nối với các bits và kích thước window đã thiết lập -đưa gói lên hàng đợi ra socket -gọi hàm tcp_transmit_skb() để tiến hành gửi gói tin, thiết lập khởi tạo kết nối TCP
tcp_v4_connect() có trong tệp net/ipv4/tcp_ipv4.c	-kiểm tra các lỗi -gọi hàm ip_route_connect() để tìm đường tới đích -tạo gói kết nối -gọi hàm tcp_connect() để gửi gói tin.
udp_close() có trong tệp net/ipv4/udp.c	-gọi hàm udp_v4_unhash() để xoá socket từ trong danh sách socket -gọi hàm destroy_sock().
udp_connect() có trong tệp net/ipv4/udp.c	-gọi hàm ip_route_connect() để tìm tuyến tới đích -cập nhật socket (địa chỉ nguồn, đích và các cổng) -thay đổi trạng thái socket ghi tuyến đích (đã thiết lập) -ghi tuyến đích trong sock->dst_cache.

4. Gửi thông điệp (Sending messages)

4.1 Tổng quan

Một thông điệp đi ra bắt đầu từ một ứng dụng gọi hàm hệ thống để ghi dữ liệu ra socket. Socket kiểm tra kiểu kết nối của nó và gọi thủ tục gửi phù hợp

(thường là INET). Hàm gửi này kiểm tra trạng thái của socket, kiểm tra kiểu giao thức của nó, và gửi dữ liệu tới thủ tục của tầng Giao vận (chẳng hạn TCP hoặc UDP). Giao thức này tạo một bộ đệm mới cho gói dữ liệu ra (một bộ đệm socket hoặc struct sk_buff skb), sao chép dữ liệu từ bộ đệm ứng dụng, và đưa thêm thông tin header của nó vào (chẳng hạn số cổng, tùy chọn, checksum) trước khi chuyển bộ đệm mới vào tầng mạng (thường là IP). Các hàm gửi IP đưa thêm các thông tin header giao thức của nó (chẳng hạn địa chỉ IP, tùy chọn, checksum). Nó cũng có thể phân mảnh gói nếu cần thiết. Tiếp theo tầng IP chuyển gói tới hàm ở tầng Liên kết, để thực hiện chuyển gói tới hàng đợi xmit của thiết bị và đảm bảo rằng thiết bị biết để truyền. Cuối cùng, thiết bị (chẳng hạn là card mạng) báo cho bus để gửi gói tin đi (xem hình 5).

4.2 Các bước gửi dữ liệu

Ghi dữ liệu vào socket

- Ghi dữ liệu vào một socket (application)
- Đưa thêm phần đầu thông điệp cho vị trí dữ liệu (socket)
- Kiểm tra các lỗi cơ bản - socket có hướng về cổng? socket có thể gửi thông điệp không? có gì sai với socket?
- Chuyển thông điệp và phần đầu tới giao thức truyền tin phù hợp (INET socket).

Tao một gói với UDP

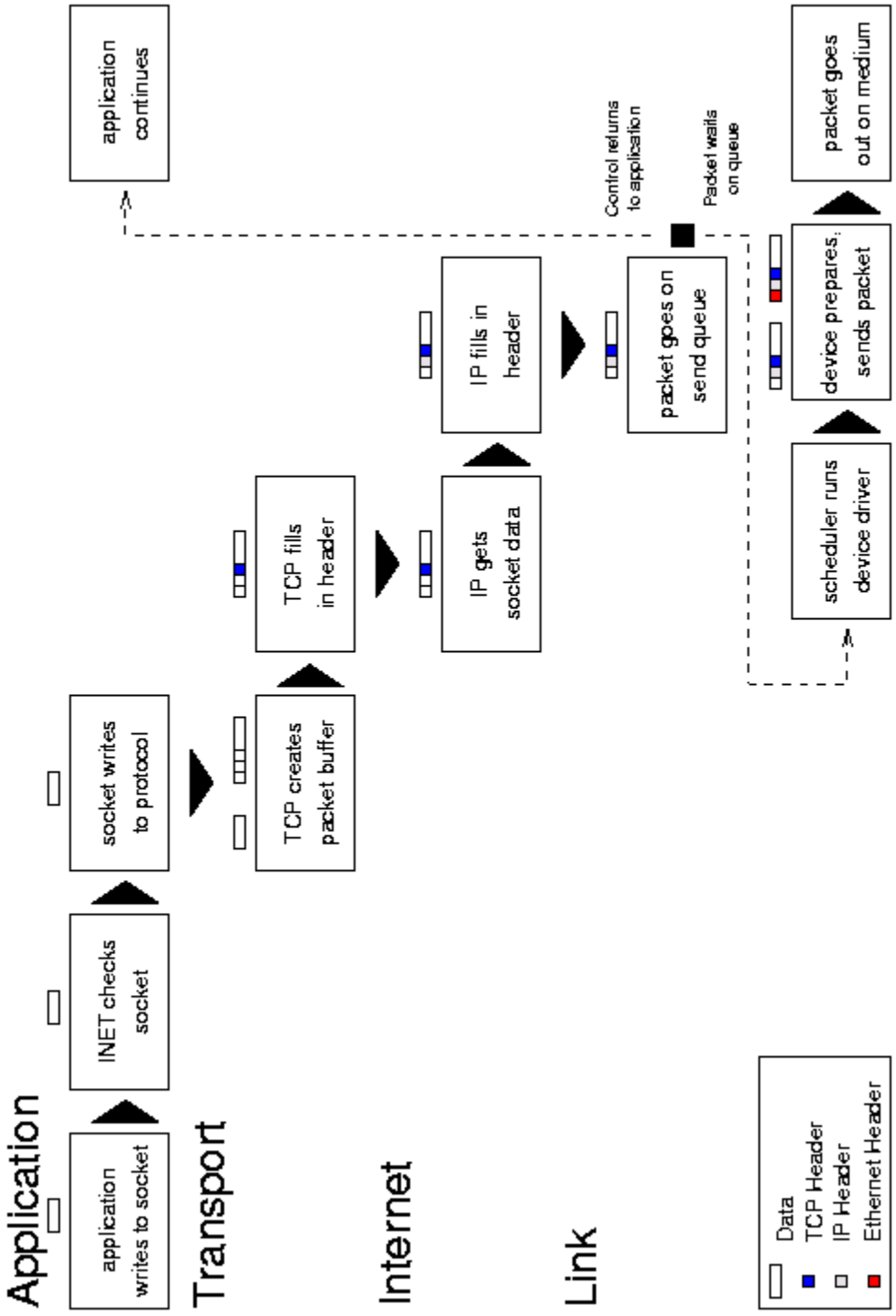
- Kiểm tra lỗi - dữ liệu có lớn quá không? Có phải là kết nối UDP?
- Bảo đảm có một đường truyền tới địa chỉ đích (gọi thủ tục định tuyến IP nếu tuyến chưa được thiết lập; lỗi nếu không có tuyến).
- Tạo phần đầu UDP (cho gói).
- Gọi hàm tạo IP và truyền.

Tao một gói với TCP

- Kiểm tra kết nối - được thiết lập chưa? đã mở chưa? socket đã làm việc chưa?
- Kiểm tra tổ hợp dữ liệu với từng phần gói nếu có thể.
- Tạo bộ đệm gói.
- Sao chép phần payload từ không gian người dùng.
- Thêm gói vào hàng đợi gửi.
- Tạo thông tin header TCP hiện hành cho gói (với ACK, SYN, ...)
- Gọi hàm truyền IP.

Boc gói trong IP

- Tạo một bộ đệm gói (nếu cần thiết - UDP).
- Tìm tuyến tới địa chỉ đích (nếu cần thiết - TCP).
- Đưa thêm phần header IP.
- Copy phần header truyền và payload từ không gian người dùng.
- Gửi gói tới hàm chuyển tới thiết bị của tuyến đích.



Hình 5: Message transmission.

Truyền một gói

- Đưa gói vào hàng đợi thiết bị ra.
- Đánh thức thiết bị.
- Đợi bộ lập lịch để chạy driver thiết bị.
- Kiểm tra thiết bị.
- Gửi phần header liên kết.
- Báo cho bus truyền gói dữ liệu qua thiết bị.

4.3 Các hàm Linux

Trong mục này chúng tôi đưa ra danh sách các hàm trong Linux kernel (theo alphabet) được coi là quan trọng trong việc truyền thông điệp (bắt đầu từ lời gọi hàm **sock_write()**), đồng thời đưa ra source code và phân tích chức năng của nó.

Functions/Tệp	Description
`dev_queue_xmit()` ở trong tệp `net/core/dev.c`	-gọi hàm `start_bh_atomic()` -nếu thiết bị có hàng đợi thì: +gọi hàm `enqueue()` để thêm gói đó vào hàng đợi +sau đó gọi hàm `qdisc_wakeup() [= qdisc_restart()]` để đánh thức thiết bị -nếu không có hàng đợi thì lần lượt gọi các hàm `hard_start_xmit()` -gọi `end_bh_atomic()`.
`DEVICE->hard_start_xmit()`ở trong tệp drivers/net/DEVICE.c	-hàm này phụ thuộc vào thiết bị, nó tiến hành kiểm tra xem phương tiện truyền được mở chưa -gửi header -báo bus để gửi gói tin -cập nhật trạng thái.
`inet_sendmsg()` ở trong tệp `net/ipv4/af_inet.c`	-thực hiện việc tách con trỏ socket sock -kiểm tra socket để chắc chắn rằng nó đang hoạt động -kiểm tra con trỏ tới giao thức -giá trị trả về của hàm là `sk->prot[tcp/udp]->`sendmsg().
`ip_build_xmit` ở trong tệp `net/ipv4/ip_output.c`	-gọi hàm `sock_alloc_send_skb()` để thiết lập bộ nhớ cho skb -khởi tạo header của skb -gọi hàm `getfrag() [= udp_getfrag()]` để copy buffer trong không gian người dùng -giá trị trả về là `rt->u.dst.output() [= dev_queue_xmit()]`.
`ip_queue_xmit()` ở trong tệp `net/ipv4/ip_output.c`	-thực hiện việc tìm tuyến -tạo IP header -phân đoạn nếu cần thiết -thêm vào IP chechsum -gọi `skb->dst->`output() [= dev_queue_xmit()].
`qdisc_restart()`ở trong tệp	-tống gói ra khỏi hàng đợi -gọi `dev->hard_start_xmit()`

net/sched/sch_generic.c	-cập nhật trạng thái -nếu có một lỗi nào đó thì đưa gói trở lại hàng đợi.
sock_sendmsg()ở trong tệp net/socket.c	-gọi hàm scm_sendmsg() [socket control message] -gọi sock->ops[inet]- >sendmsg(), và huỷ bỏ scm.
sock_write() ở trong tệp net/socket.c	-gọi hàm socki_lookup() để kết hợp socket với fd/file inode -tạo và điền vào header của thông báo với dữ liệu về kích thước/địa chỉ -giá trị trả về là hàm sock_sendmsg()
tcp_do_sendmsg() ở trong tệp net/ipv4/tcp.c	-chờ kết nối, nếu cần thiết -gọi skb_tailroom() và thêm data vào gói đang đợi nếu có thể -kiểm tra trạng thái của window -gọi hàm sock_wmalloc() để tạo bộ nhớ cho skb -gọi hàm csum_and_copy_from_user() để copy gói và tổng kiểm tra -cuối cùng là gọi hàm tcp_send_skb().
tcp_send_skb()ở trong tệp net/ipv4/tcp_output.c	-gọi __skb_queue_tail() để thêm gói vào trong hàng đợi -gọi hàm tcp_transmit_skb() nếu có thể.
tcp_transmit_skb()ở trong tệp net/ipv4/tcp_output.c	-tạo TCP header và thêm vào tổng kiểm tra -gọi hàm tcp_build_and_update_options() -kiểm tra các ACK và SYN -gọi tp->af_specific[ip]- >queue_xmit().
tcp_v4_sendmsg() ở trong tệp net/ipv4/tcp_ipv4.c	-kiểm tra kiểu địa chỉ IP,mở kết nối, các địa chỉ cổng -giá trị trả về là hàm tcp_do_sendmsg()
udp_getfrag() ở trong tệp net/ipv4/udp.c	-copy và tổng kiểm tra một buffer từ không gian người dùng đến
udp_sendmsg() ở trong tệp net/ipv4/udp.c	-kiểm tra độ dài, cờ, giao thức -thiết lập UDP header và thông tin địa chỉ -kiểm tra multicast -điền vào bảng định tuyến -điền phần còn lại của header -gọi hàm ip_build_xmit() -cập nhật trạng thái của UDP -giá trị trả về của hàm là err.

5. Nhận thông điệp

5.1 Tổng quan

Một thông điệp được gửi đến với một ngắt để báo hiệu, khi hệ thống được thiết bị báo là thông điệp đã sẵn sàng. Thiết bị cấp phát không gian bộ nhớ và báo cho bus đưa thông điệp vào không gian đó. Sau đó nó chuyển gói tin tới tầng liên

kết, đưa vào hàng đợi backlog, và đánh dấu cờ network để "bottom-half" tiếp theo chạy.

Bottom-half là một hệ thống Linux mà tối thiểu khối lượng công việc thực hiện trong khi ngắt. Thực hiện nhiều tiến trình trong một ngắt là không chính xác vì nó ngắt một tiến trình đang chạy; thay vào đó, một trình điều khiển ngắt có "top-half" và một "bottom-half". Khi một ngắt đến, top-half chạy và chú ý tới bất kỳ thao tác đặc biệt nào, chẳng hạn chuyển dữ liệu từ hàng đợi của thiết bị vào bộ nhớ của nhân. Sau đó nó đánh dấu một cờ để báo cho nhân rằng có nhiều công việc phải làm (khi nhân có thời gian), rồi quay lại kiểm soát tiến trình của mình. Lần sau, khi process scheduler chạy, nó thấy cờ, thực hiện thêm công việc, và chỉ lập lịch trình bất kỳ tiến trình thông thường nào.

Khi tiến trình bộ lập lịch thấy rằng có các tác vụ mạng cần thực hiện, nó chạy bottom-half mạng. Hàm này lấy các gói từ hàng đợi backlog, đưa chúng tới giao thức đã biết (thường là IP), và chuyển chúng tới hàm nhận của giao thức đó. Tầng IP kiểm tra gói xem có lỗi không và chuyển nó; gói sẽ được đưa ra hàng đợi ra (nếu gói tin cho máy khác) hoặc đẩy lên tầng Giao vận (TCP hoặc UDP). Tầng này lại kiểm tra lỗi, tìm kiếm socket được gán với cổng được chỉ ra trong gói, và đưa gói vào cuối hàng đợi nhận của socket.

Khi gói nằm trong hàng đợi của socket, socket sẽ đánh thức tiến trình ứng dụng mà sử dụng nó (nếu cần thiết). Tiến trình đó có thể tạo hoặc nhận từ lời gọi read của hệ thống mà copy dữ liệu từ gói trong hàng đợi vào bộ đệm của nó (xem hình 6).

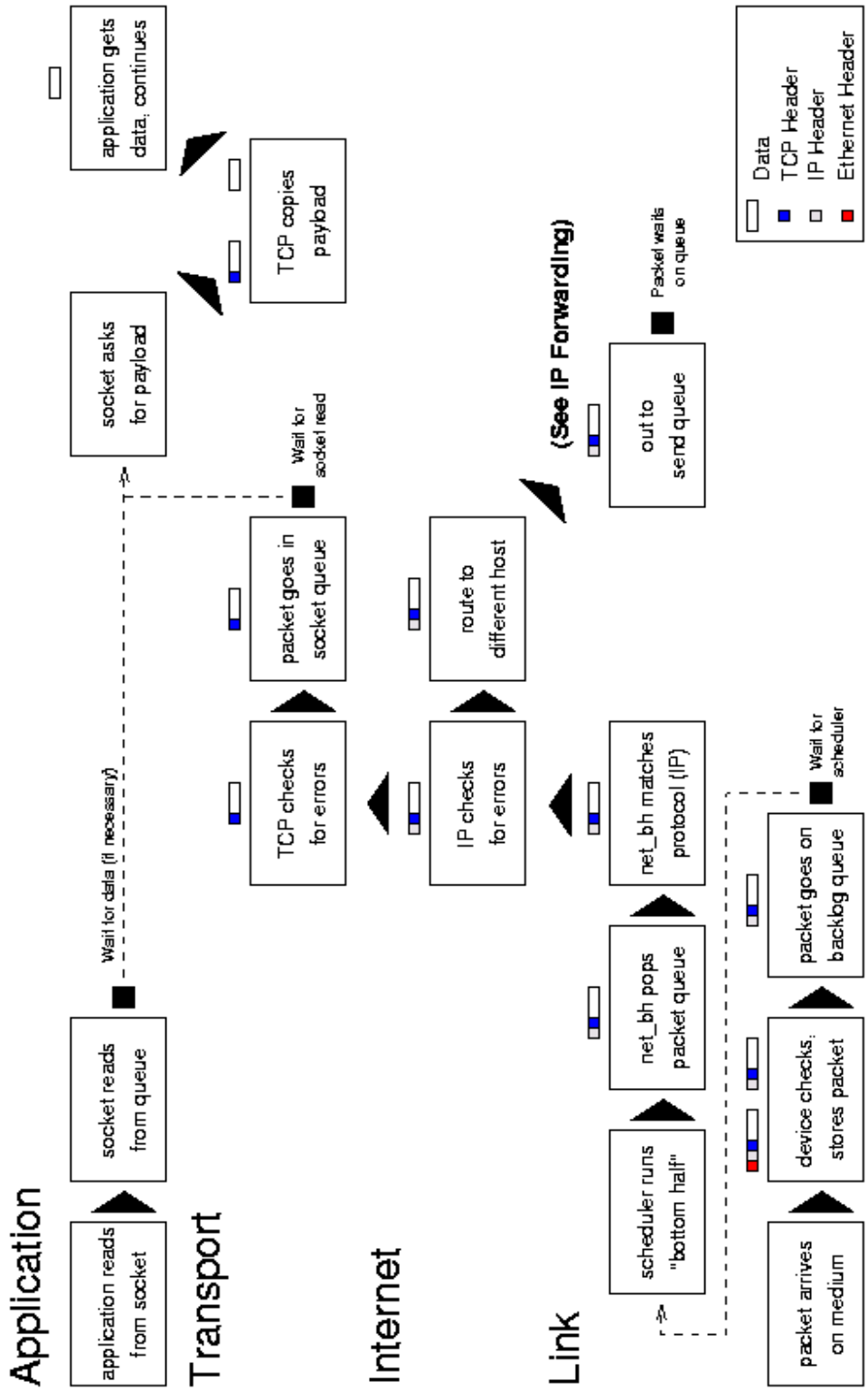
5.2 Các bước nhận dữ liệu

Đọc dữ liệu từ socket

- Cố đọc dữ liệu từ socket (application)
- Đưa thêm phần header của thông điệp với vị trí của bộ đệm (socket).
- Kiểm tra các lỗi cơ bản - socket có hướng về cổng? socket có thể gửi thông điệp không? có gì sai với socket?
- Chuyển thông điệp và phần đầu tới giao thức truyền tin phù hợp (INET socket).
- Đợi cho đến khi có đủ dữ liệu để đọc từ socket (TCP/UDP)

Nhận một gói

- Đánh thức thiết bị nhận (interrupt)
- Kiểm tra thiết bị (device)
- Nhận header liên kết.
- Cấp phát không gian cho gói.
- Báo cho bus để đưa gói vào bộ đệm.
- Đưa gói lên hàng đợi backlog
- Thiết lập cờ để chạy bottom half mạng khi khi có thể.
- Trả quyền điều khiển cho tiến trình hiện hành.



Hình 6: Receiving messages.

Chạy "Bottom Half" mạng

- Chạy bottom half mạng (schedule)
- Gửi các gói bất kỳ đang đợi để giải quyết các ngắt.
- Lặp tất cả các gói trong hàng đợi và chuyển gói lên giao thức nhận Internet - IP.
- Làm sạch (giải phóng) hàng đợi gửi
- Thoát khỏi bottom half.

Hủy phần bọc một gói trong IP

- Kiểm tra các lỗi gói - quá ngắn không? có quá dài không? phiên bản không đúng? lỗi checksum?
- Ghép các phần của gói lại nếu cần thiết.
- Lấy định tuyến cho gói (có thể cho máy này hoặc có thể cần để chuyển).
- Gửi gói tới thủ tục điều khiển đích (TCP hoặc UDP hoặc có thể truyền lại tới một máy khác).

Chấp nhận một gói trong UDP

- Kiểm tra phần đầu của UDP xem có lỗi không?
- Kiểm tra địa chỉ đích cho socket
- Gửi một thông báo lỗi trở lại nếu không có socket như vậy.
- Đưa gói vào hàng đợi nhận của socket phù hợp.
- Đánh thức tiến trình đang đợi tiến trình từ socket đó.

Chấp nhận một gói trong TCP

- Kiểm tra các cờ; lưu giữ gói trong không gian đúng nếu có thể.
- Nếu đã nhận, gửi tín hiệu ACK ngay lập tức và bỏ gói.
- Xác định packet thuộc về gói nào.
- Đưa gói vào hàng đợi nhận của socket phù hợp.
- Đánh thức và tiến trình đang đợi dữ liệu từ socket đó.

Đọc từ socket (phần II)

- Đánh thức khi dữ liệu sẵn sàng (socket)
- Gọi hàm nhận của tầng Giao vận.
- Chuyển dữ liệu từ hàng đợi nhận tới bộ đệm người dùng (TCP/UDP).
- Trả lại dữ liệu và điều khiển cho ứng dụng (socket).

5.3 Các hàm trong Linux

Trong mục này chúng tôi chỉ ra danh sách các hàm của Linux kernel được coi là quan trọng trong việc nhận thông điệp, đồng thời phân tích cách làm việc của source code. Các lời gọi hàm thiết lập mạng bắt đầu từ hàm **DEVICE_rx**(), và các lời gọi hàm ngắt ứng dụng bắt đầu từ hàm **sock_read**().

Functions/File	Descriptions
DEVICE_rx() có trong tệp drivers/net/DEVICE.c	-hàm này phụ thuộc vào thiết bị (lấy điều khiển từ ngắt) -thực hiện kiểm tra trạng thái để chắc chắn là nó đang nhận -gọi hàm dev_alloc_skb() để dành ra một không gian nhớ cho gói tin -lấy gói tin từ bus hệ thống -gọi hàm dev_alloc_skb() để chỉ ra kiểu giao thức -gọi hàm netif_rx() –cập nhật trạng thái của card, trả về ngắt.
inet_recvmsg() có trong tệp net/ipv4/af_inet.c	-trích con trỏ socket *sock* -kiểm tra socket để chắc chắn rằng nó đã chấp nhận -kiểm tra con trỏ tới giao thức -giá trị trả về của hàm là sk->prot[tcp/udp]- >recvmsg().
ip_rcv() có trong tệp net/ipv4/ip_input.c	-thực hiện việc kiểm tra lỗi cho gói tin: +sai về độ dài (quá ngắn hoặc quá dài) +sai về phiên bản (không phải ipv4) +sai về giá trị tổng kiểm tra -gọi hàm __skb_trim() để xoá bộ đệm -tách gói tin (defrags packet) nếu cần thiết -gọi hàm ip_route_input() để định tuyến cho gói tin -kiểm tra và điều khiển các tuỳ chọn IP -giá trị trả về của hàm là skb->dst->input() [= tcp_rcv,udp_rcv()].
net_bh() có trong tệp net/core/dev.c	-hàm này thực thi bởi trình lập lịch -nếu có nhiều gói tin đang đợi đi ra thì gọi hàm qdisc_run_queues() (xem phần gửi thông điệp) -trong khi hàng đợi backlog mà không rỗng thì thực hiện: +cho phép nửa dưới thực thi +gọi hàm skb_dequeue() để lấy gói tiếp theo +nếu gói tin cho nhiều người thì đưa nó lên hàng đợi gửi (FASTROUTED) +lặp trong các danh sách giao thức đến khi tìm được đúng kiểu của giao thức +gọi hàm pt_prev->func() [= ip_rcv()] để đưa gói tin tới giao thức tương ứng -sau đó gọi hàm qdisc_run_queues() để đưa tới đầu ra (nếu cần).
netif_rx() có trong tệp net/core/dev.c	-đưa thời gian vào trong skb->stamp –nếu hàng đợi backlog quá đầy thì ngừng gói tin lại -nếu không: + gọi hàm skb_queue_tail() để đưa gói tin vào trong hàng đợi backlog +đánh dấu cho lần thực thi sau.
sock_def_readable() có trong tệp net/core/sock.c	-gọi hàm wake_up_interruptible() để đưa các tiến trình đang đợi sử lý lên hàng đợi thực thi -gọi hàm sock_wake_async() để gửi SIGIO tới tiến trình socket.

sock_queue_rcv_skb() có trong tệp include/net/sock.h	-gọi hàm skb_queue_tail() để đẩy gói tin vào trong hàng đợi nhận socket -cuối cùng là gọi hàm sk->data_ready() [= sock_def_readable()].
sock_read() có trong tệp net/socket.c	-thiết lập các header của thông báo -giá trị trả về là kết quả đọc được là hàm sock_recvmsg().
sock_recvmsg() có trong tệp net/socket.c	-đọc scm (socket management packet) hoặc gói tin qua lời gọi hàm sock->ops[inet]- >recvmsg().
tcp_data() có trong tệp net/ipv4/tcp_input.c	-rút ngắn hàng đợi nhận (nếu cần) -gọi hàm tcp_data_queue() để đưa gói ra hàng đợi -gọi sk->data_ready() để đánh thức socket.
tcp_data_queue() có trong tệp net/ipv4/tcp_input.c	-kiểm tra nếu gói tin ở ngoài dãy trong hàng đợi thì: +nếu nó đã cũ thì huỷ bỏ ngay lập tức +nếu không thì lưu nó vào vị trí tương ứng -sau đó gọi hàm __skb_queue_tail() để đưa gói tin lên hàng đợi nhận của socket -cuối cùng nó làm nhiệm vụ cập nhật trạng thái kết nối.
tcp_rcv_established() có trong tệp net/ipv4/tcp_input.c	-hàm này thực hiện việc kiểm tra nếu là fast path thì: +kiểm tra tất cả các cờ và thông tin về header +gửi ACK +sau đó gọi hàm _skb_queue_tail() để đưa gói tin vào hàng đợi nhận của socket -nếu là slow path thì: +nếu không nằm trong dãy hàng đợi thì gửi ACK và bỏ gói đó +kiểm tra các tín hiệu FIN, SYN, RST, ACK +gọi hàm tcp_data() để đưa gói tin vào hàng đợi +cuối cùng là gửi ACK
tcp_recvmsg() có trong tệp net/ipv4/tcp.c	-tiến hành kiểm tra các lỗi -chờ cho đến khi có tối thiểu một gói tin -sau đó xoá socket nếu kết nối đã đóng -gọi hàm memcpy_toiovec() để copy payload từ bộ đệm socket vào trong không gian người dùng -gọi hàm cleanup_rbuf() để giải phóng bộ nhớ và gửi ACK nếu cần thiết -cuối cùng gọi hàm remove_wait_queue() để đánh thức tiến trình (nếu cần).
udp_queue_rcv_skb() có trong tệp net/ipv4/udp.c	-gọi hàm sock_queue_rcv_skb() -cập nhật trạng thái UDP (xoá skb nếu hàng đợi bị lỗi).
udp_rcv() có trong tệp net/ipv4/udp.c	-lấy UDP header, lọc gói tin, kiểm tra checksum (nếu cần) -kiểm tra multicast -gọi hàm udp_v4_lookup() để tìm đúng gói tin cho socket -nếu không tìm thấy socket thì gửi thông báo ICMP trở lại đồng thời giải phóng skb và ngừng -cuối cùng nó gọi hàm udp_deliver() [= udp_queue_rcv_skb()]

`udp_recvmsg()` có trong tệp `net/ipv4/udp.c`	-gọi hàm `skb_recv_datagram()` để lấy gói tin trong hàng đợi -gọi hàm `skb_copy_datagram_iovec()` để chuyển payload từ trong bộ đệm của socket vào trong không gian người dùng -cập nhật thời gian của socket điền thông tin về nguồn tin vào header của thông điệp -việc cuối cùng là giải phóng bộ nhớ của gói tin.

6. IP Forwarding

Phần này chỉ trình bày về mặt định tuyến (bởi IP forwarding) trong truyền thông thông điệp. Nó cung cấp một cách tổng quát về tiến trình, kiểm tra các gói được chuyển qua các tầng, chi tiết về các tác động của từng tầng, và tổng quát hóa các mã nguồn trong nhân.

6.1 Tổng quan

Hình dưới là một sơ đồ tổng quát tiến trình forwarding, nó cũng là sự tổ hợp của các quá trình gửi và nhận.

Một gói tin được chuyển đến sẽ với một ngắt, khi hệ thống được báo rằng thiết bị có một thông điệp đang chờ. Thiết bị cấp phát không gian lưu trữ và báo cho bus đưa thông điệp vào không gian đó. Sau đó nó chuyển gói tới tầng liên kết, đưa nó lên hàng đợi backlog, đánh dấu cờ network để "bottom-half" tiếp theo chạy, và trả điều khiển cho tiến trình hiện hành.

Khi tiến trình lập lịch tiếp theo chạy, nó thấy rằng có tác vụ mạng cần thực hiện và chạy "bottom-half" mạng. Hàm này lấy các gói khỏi hàng đợi backlog, đưa chúng lên IP và chuyển chúng tới hàm nhận. Tầng IP kiểm tra gói xem có lỗi, và định tuyến nó; gói sẽ được chuyển lên tầng Giao vận (TCP hoặc UDP nếu gói tin được gửi cho máy này) hoặc hướng hàm IP forwarding. Trong hàm forwarding, IP kiểm tra gói và gửi một thông điệp ICMP về cho người gửi nếu có gì đó sai. Sau đó nó copy gói vào một bộ đệm mới và chia nhỏ chúng nếu cần thiết.

Cuối cùng tầng IP chuyển gói xuống hàm ở tầng liên kết, mà chuyển gói vào hàng đợi xmit của thiết bị gửi và đảm bảo thiết bị biết rằng nó cần truyền dữ liệu đi. Cuối cùng thiết bị (chẳng hạn card mạng) báo cho bus để gửi gói dữ liệu đi.

Hình 7: IP forwarding.

6.2 Các bước chuyển IP

Nhận một gói

- Đánh thức thiết bị nhận (interrupt)
- Kiểm tra thiết bị (device)
- Nhận phần header liên kết
- Cấp phát không gian cho gói
- Báo cho bus đưa gói vào bộ đệm
- Đưa gói vào hàng đợi backlog
- Thiết lập cờ để chạy bottom half mạng khi có thể.
- Trả điều khiển về cho tiến trình hiện hành.

Chạy "Bottom Half"

- Chạy network bottom half (schedule)
- Gửi các gói đang chờ giải quyết ngắt (net_bh)
- Lặp tất cả các gói trong hàng đợi backlog và chuyển gói lên giao thức chấp nhận Internet (IP).
- Làm sạch hàng đợi gửi lại một lần nữa.
- Thoát khỏi bottom half.

Kiểm tra gói trong IP

- Kiểm tra lỗi trong gói - xem có quá ngắn không? quá dài không? đúng phiên bản không? checksum có lỗi không?
- Chia nhỏ gói nếu cần thiết.
- Lấy định tuyến cho gói (có thể cho máy này hoặc có thể cần chuyển)
- Gửi gói tới thủ tục điều khiển đích (truyền lại tới một máy khác)

Chuyển gói trong IP

- Kiểm tra trường TTL (giảm nó đi)
- Kiểm tra gói không đúng cho định tuyến
- Gửi ICMP trở lại cho người gửi nếu có lỗi.
- Copy gói vào bộ đệm mới và giải phóng bộ đệm cũ.
- Thiết lập các tùy chọn IP.
- Chia nhỏ gói nếu nó quá lớn cho đích mới.
- Gửi gói tới hàm ra thiết bị.

Truyền một gói

- Đưa gói lên hàng đợi ra của thiết bị.
- Đánh thức thiết bị.
- Đợi bộ lập lịch (scheduler) chạy driver của thiết bị.
- Kiểm tra thiết bị.
- Gửi phần đầu (header) của liên kết.

- Báo cho bus để truyền gói lên thiết bị.

6.3 Các hàm trong Linux

Trong mục này chúng tôi đưa ra danh sách (theo alphabe) các hàm trong Linux kernel được coi là quan trọng trong việc IP forwarding, đồng thời phân tích hoạt động của source code. Các lời gọi hàm bắt đầu từ hàm **DEVICE_rx()**.

Functions/File	Descriptions
`dev_queue_xmit()` có trong tệp `net/core/dev.c`	-gọi hàm `start_bh_atomic()` -nếu thiết bị có một hàng đợi thì: +gọi hàm `enqueue()` để thêm gói tin vào hàng đợi +gọi hàm `qdisc_wakeup() [= qdisc_restart()]` để đánh thức thiết bị -nếu không có hàng đợi gọi hàm `hard_start_xmit()` -gọi hàm `end_bh_atomic()`.
`DEVICE->hard_start_xmit()` có trong tệp drivers/net/DEVICE.c	-hàm này phụ thuộc vào thiết bị -nó thực hiện kiểm tra phương tiện truyền xem có mở không -gửi header -báo cho bus gửi gói tin -cuối cùng nó cập nhật trạng thái.
`>>> DEVICE_rx()` có trong tệp `drivers/net/DEVICE.c`	-hàm này phụ thuộc vào thiết bị (lấy điều khiển từ ngắt) -nó thực hiện kiểm tra trạng thái để chắc chắn là ở trạng thái nhận -gọi hàm `dev_alloc_skb()` để dự trữ không gian nhớ cho gói tin -lấy gói tin từ bus hệ thống -gọi hàm `eth_type_trans()` để xác định kiểu giao thức -gọi hàm `netif_rx()` -cập nhật trạng thái card (trả về từ ngắt).
`ip_finish_output()` có trong tệp `include/net/ip.h`	-thiết lập thiết bị nhận của thiết bị ra chuẩn nhằm mục đích lấy thông tin về tuyến -gọi hàm đưa gói tin cho đích `[= dev_queue_xmit()]`.
`ip_forward()` có trong tệp `net/ipv4/ip_forward.c`	-kiểm tra router alert -nếu gói tin là không của máy nào thì huỷ nó -nếu TTL đã hết hạn thì bỏ gói tin đó và gửi thông báo ICMP trở lại -nếu tuyến ngắn nhất không có thì bỏ gói tin và gửi thông báo ICMP trở lại người gửi -nếu cần thiết thì gửi thông báo ICMP báo cho gói tin đã chuyển hướng -copy và giải phóng gói tin cũ -giảm TTL -nếu có một vài tuỳ chọn, gọi hàm `ip_forward_options()` để thiết lập chúng -cuối cùng nó gọi hàm `ip_send()`

ip_rcv() có trong tệp net/ipv4/ip_input.c	-hàm này tiến hành kiểm tra các lỗi của gói tin: +lỗi về độ dài (quá ngắn hoặc quá dài) +lỗi về phiên bản (không phải ipv4) +lỗi về checksum -sau đó gọi hàm __skb_trim() để xoá bộ đệm -ghép gói tin nếu cần -gọi hàm ip_route_input() đưa gói tin ra tuyến -kiểm tra và điều khiển các tuỳ chọn của IP -giá trị trả về của hàm là skb->dst->input() [= ip_forward()].
ip_route_input() có trong tệp net/ipv4/route.c	-gọi hàm rt_hash_code() để lấy thứ tự của bảng định tuyến -vòng lặp trong bảng định tuyến với giá trị bắt đầu là hash cho đến khi tìm tuyến cho gói tin -nếu tìm thấy thì: +cập nhật trạng thái tuyến (thời gian và sử dụng) +thiết lập đích của gói tin vào trong entry của bảng định tuyến + trả về thành công -nếu không tìm thấy thì: +kiểm tra có phải là các địa chỉ multicast +trả về hàm ip_route_input_slow()
ip_route_output_slow() có trong tệp net/ipv4/route.c	-nếu địa chỉ nguồn đã biết thì tìm kiếm thiết bị ra chuẩn -nếu địa chỉ đích đã biết thì thiết lập loopback -sau đó gọi hàm fib_lookup() để tìm tuyến -cấp phát bộ nhớ mới cho entry bảng định tuyến -khởi tạo entry cho bảng với địa chỉ nguồn, đích, TOS , thiết bị ra và các cờ -gọi hàm rt_set_nexthop() để tìm đích tiếp theo -giá trị trả về hàm rt_intern_hash().
ip_send() có trong tệp include/net/ip.h	-gọi hàm: ip_fragment() nếu gói tin quá lớn so với thiết bị -gọi hàm ip_finish_output().
net_bh() có trong tệp net/core/dev.c	-hàm này được chạy bởi trình lập lịch -nó tiến hành kiểm tra nếu có nhiều gói tin đang đợi ở đầu ra thì gọi hàm qdisc_run_queues() (xem phần gửi thông điệp) -trong khi hàng đợi của backlog không rỗng thì: +cho phép nửa dưới thực thi +gọi hàm skb_dequeue() để nhận gói tiếp theo +nếu là gói tin thì cho một số người khác nếu không (FASTROUTED) thì đưa nó lên hàng đợi gửi +tiến hành vòng lặp trong các danh sách giao thức để tìm đúng kiểu của giao thức +sau đó gọi hàm pt_prev->func() [= ip_rcv()] để đưa gói tin tới giao thức tương ứng -cuối cùng hàm này gọi hàm qdisc_run_queues() để đưa ra ngoài.

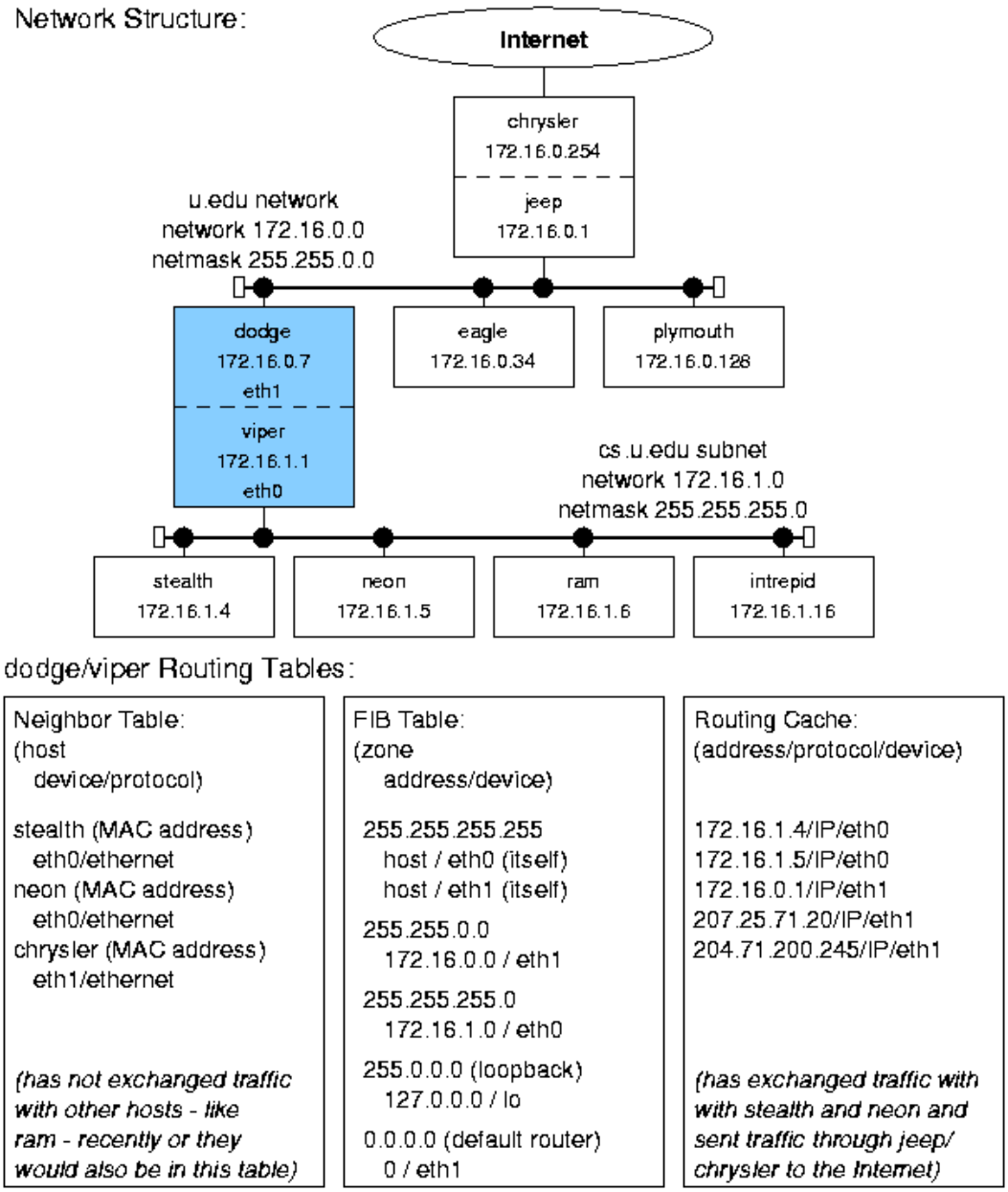
`netif_rx()` có trong tệp `net/core/dev.c`	-đưa thời gian vào `skb->stamp` -nếu hàng đợi của backlog đã quá đầy thì huỷ bỏ gói tin -nếu không thì: + gọi hàm `skb_queue_tail()` để đưa gói tin vào hàng đợi backlog +đánh dấu tất cả lại cho lần thực thi sau.
`qdisc_restart()` có trong tệp `net/sched/sch_generic.c`	-đẩy gói tin ra hàng đợi -gọi hàm `dev->hard_start_xmit()` -cập nhật trạng thái -nếu có một lỗi nào đó thì đưa gói tin trở lại hàng đợi
`rt_intern_hash()` có trong tệp `net/ipv4/route.c`	-đưa tuyến mới vào trong bảng định tuyến

7. Internet Protocol Routing

Phần này trình bày những vấn đề cơ bản về IP Routing: tổng quan về cách làm việc của việc định tuyến, kiểm tra cách mà bảng định tuyến (routing tables) được thiết lập và cập nhật, cuối cùng là tóm lược về mã nguồn định tuyến trong kernel.

7.1 Tổng quan

Linux chứa 3 tập dữ liệu định tuyến sau - một cho các máy tính được kết nối trực tiếp tới đó là **Neighbor Table** (ví dụ: trong mạng LAN), và hai cho các máy tính được kết nối gián tiếp (qua mạng IP): **Forwarding Information Base** và **Routing Cache.**



Hình 8: General routing table example.

Bảng neighbor table chứa thông tin địa chỉ các máy được kết nối vật lý tới nó (neighbor). Bảng này bao gồm thông tin về thiết bị kết nối tới neighbor và những giao thức nào sử dụng để trao đổi dữ liệu. Linux sử dụng ARP (Address Resolution Protocol) để duy trì và cập nhật bảng này; nó được thay đổi khi cần thiết thậm trí biến mất nếu không được sử dụng trong thời gian đó.

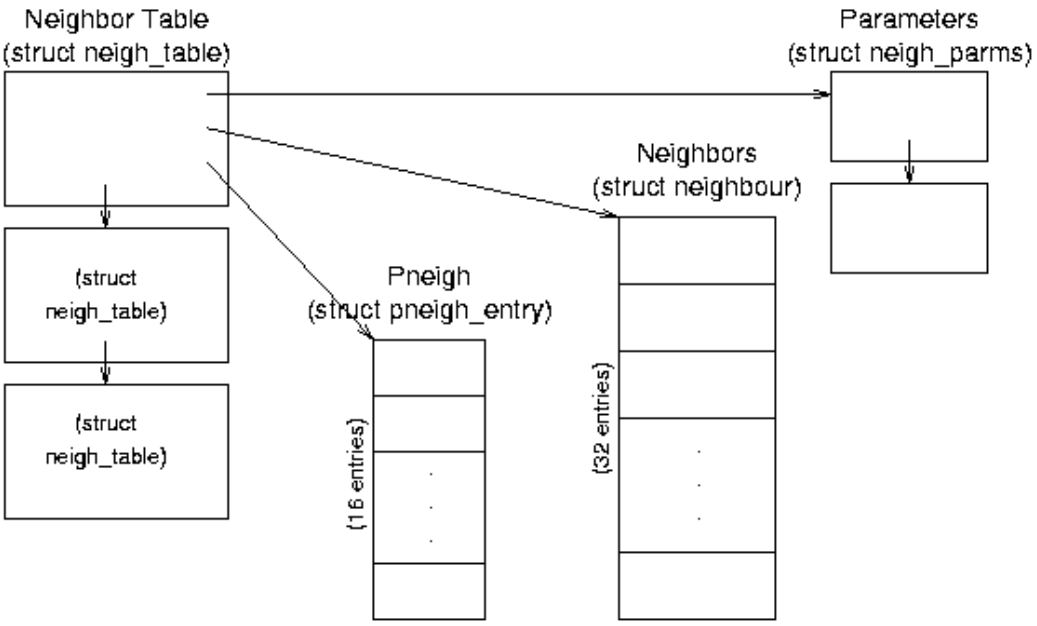
Linux sử dụng hai bảng định tuyến (tương đối phức tạp) để duy trì các địa chỉ IP: một với mục đích FIB (Forwarding Information Base) chỉ tới tất cả các địa chỉ có thể và một routing cache (bộ đệm định tuyến) nhỏ hơn (với mục đính làm nhanh hơn) với dữ liệu định tuyến được sử dụng. Khi một gói IP (IP packet) cần đi tới một máy ở xa, việc đầu tiên của tầng IP (IP layer) là kiểm tra routing cache cho entry mà tương ứng với nguồn, đích và kiểu của dịch vụ. Nếu đúng với entry thì nó được sử dụng. Nếu không phù hợp thì IP truy vấn thông tin định tuyến trong FIB (điều này sẽ làm cho việc tìm tuyến lâu hơn), và tạo một cache entry mới với dữ liệu vừa tìm được và sau đó sử dụng cache entry này.

7.2 Bảng định tuyến (Routing Tables)

Trong các bảng định tuyến có các biến với các kiểu như **u32** (host byte order) và **_u32** (network byte order). Trên kiến trúc của Intel chúng tương đương với **unsigned int** và trong con trỏ hàm được dịch ra băng cách gọi hàm **ntohl()**.

Neighbor Table

Như đã đề cập ở trên thì Neighbor Table chứa thông tin về các máy được kết nối tới nó. Các entry có thể thay đổi, nên bảng này có thể không có các entry (nếu hiện tại máy không được giao thông trên mạng) hoặc có thể có các entry chứa thông các máy đã kết nối vật lý tới mạng của nó. Entry trong bảng chứa thông tin về địa chỉ, thiết bị, giao thức.



Hình 10: Quan hệ về cấu trúc dữ liệu trong bảng Neighbor Table.

struct neigh_table *neigh_tables: biến toàn cục này là một con trỏ đến danh sách của **neighbor tables**; từng bảng chứa tập các hàm chung và dữ liệu và

một bảng **hash table** chỉ thông tin về tập các neighbor. Cấu trúc này rất chi tiết với những bảng chứa thông tin như: thời gian truyền thông của các messages, kích thước hàng đợi (queue size), các con trỏ thiết bị, và các con trỏ tới hàm thiết bị.

Neighbor Table Structure (struct neigh_table) là cấu trúc các thông tin neighbor thông thường và bảng dữ liệu neighbor và dữ liệu pneigh (pneigh data). Tất cả các máy đã kết nối theo kiểu "kết nối đơn" (một card Ethernet) sẽ có bảng tương tự.

- Struct neigh_table *next: trỏ tới bảng tiếp theo của danh sách.
- Struct neigh_parms parms: cấu trúc bao gồm thông báo về thời gian sống, độ lớn hàng đợi, và thông tin về thống kê; (phần này ở đầu của danh sách).
- Struct neigh_parms *parms_list: trỏ tới một danh sách các cấu trúc tin.
- Struct neighbour *hash_buckets[]: bảng hash table của các neighbor đã kết hợp với bảng này (có NEIGH-HASHMASK+1 (32) buckets).
- Struct pneigh_entry *phash_buckets[]: bảng hash table các cấu trúc con trỏ thiết bị và các khoá (có PNEIGH_HASHMASSK+1 (16) buckets).
- Một số các trường khác bao gồm thông tin về điều khiển thời gian, con trỏ hàm, khoá và thống kê.

Cấu trúc dữ liệu Neighbor (`struct neighbour`) - cấu trúc này chứa thông tin về từng neighbor. Nó bao gồm những cấu trúc sau:

- struct device *dev - con trỏ tới thiết bị được kết nối tới neighbor này.
- __u8 nud_state - các cờ trạng tháip; các giá trị : chưa hoàn thành, đã hoàn thành, trạng thái, ...; cũng bao gồm cả thông tin không đổi và sử dụng ARP.
- struct hh_cache *hh - con trỏ tới header phần cứng trong cache (hardware header) phục vụ cho việc truyền thông tới neighbor này.
- struct sk_buff_head arp_queue - con trỏ tới các gói ARP cho neighbor này.
- Một số trường khác gồm các con trỏ tới danh sách, con trỏ hàm, và thống kê một số thông tin khác.

Thông tin đinh tuyến (FIB)

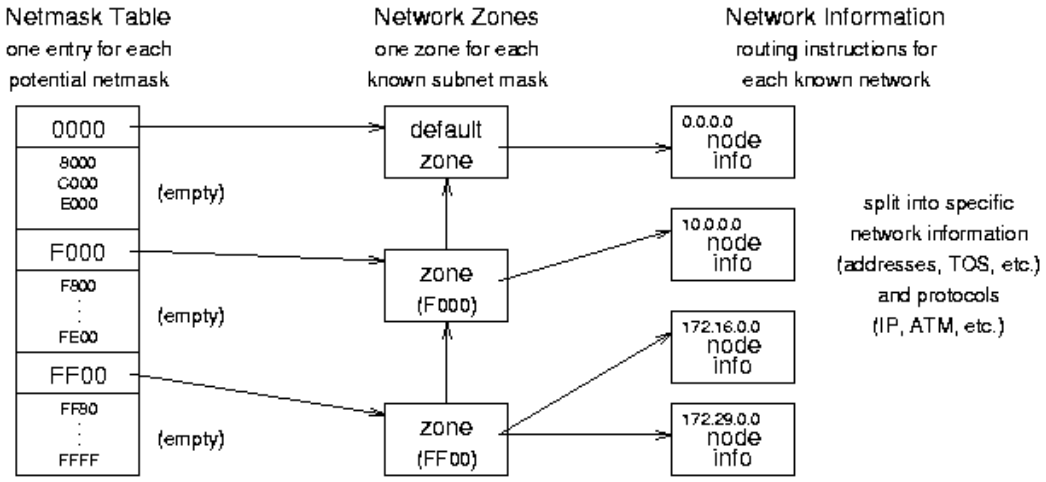


Figure 11: Forwarding Information Base (FIB) conceptual organization.

Forwarding Information Base là cấu trúc định tuyến quan trọng trong kernel Linux; nó là một cấu trúc phức tạp, bao gồm thông tin định tuyến cần thiết cho tìm kiếm đúng địa chỉ IP thông qua mặt nạ mạng (network mask). Nó là một bảng lớn với thông tin chung của các địa chỉ ở trên bảng và thông tin định danh ở sau cuối bảng. Tầng IP đưa vào bảng địa chỉ đích của gói và so sánh nó với netmask để phân biệt đích của gói tin. Nếu không phải, thì IP lại tìm đến netmask tiếp theo và tiếp tục như vậy cho đến khi đúng, IP sẽ copy trực tiếp máy ở xa vào trong routing cache và gửi gói tin theo đường này. Các hình 12 và 13 là cách tổ chức và cấu trúc dữ liệu được sử dụng trong FIB.

`struct fib_table *local_table, *main_table` - đây là các biến toàn cục là các con trỏ truy cập tới bảng FIB; chúng trỏ tới cấu trúc bảng trỏ tới các hash table, các bảng hash này lại trỏ tới các vùng tương ứng. Nội dung của biến main_table có trong variable are in */proc/net/route*.

Cấu trúc `fib_table` (FIB Table Structure) - *include/net/ip_fib.h* - các cấu trúc này gồm các bảng nhảy hàm (function jump) và từng bảng này trỏ tới một bảng hash (chứa thông tin vùng).

- `int (*tb_functions)()` - các con trỏ tới các hàm (lookup, delete, insert, ...) chúng được khởi tạo bằng hàm `fn_hash_function()`.
- `unsigned char tb_data[0]` - con trỏ tới bảng FIB hash table (mặc dù khai báo của nó là một mảng kí tự (character array).
- `unsigned char tb_id` - định danh của bảng (table identifier); 255 cho bảng `local_table`, 254 cho bảng `main_table`.
- `unsigned tb_stamp`

Cấu trúc của Netmask Table là `fn_hash` - trong tệp *net/ipv4/fib_hash.c* - cấu trúc này bao gồm các con trỏ tới từng vùng riêng biệt (được tổ chức bởi netmask).

Sẽ có một netmask cho từng bảng FIB, trừ khi 2 bảng cùng trỏ tới một bảng hash table.

- `struct fn_zone *fn_zones[33]` - là các con trỏ tới các zone entry (mỗi một vùng tương ứng với một bit trong mask; `fn_zone[0]` trỏ tới vùng có netmask 0x0000, `fn_zone[1]` trỏ tới vùng có netmask 0x8000, và `fn_zone[32]` trỏ tới vùng có netmask 0xFFFF.
- `struct fn_zone *fn_zone_list` - con trỏ tới vùng đầu tiên (không trống) trong danh sách; nếu có một entry cho netmask 0xFFFF nó sẽ trỏ tới vùng đó, nếu không nó có thể trỏ tới vùng 0xFFF0 hoặc 0xFF00 hoặc 0xF000, ...

Cấu trúc Network Zone **fn_zone** - trong tệp *net/ipv4/fib_hash.c* - các cấu trúc này bao gồm một số thông tin hash và các con trỏ tới hash table của các node. Sẽ có một Network Zone cho từng netmask đã biết.

- `struct fn_zone *fz_next` - con trỏ tới vùng không trống tiếp theo trong cấu trúc hash (`fn_hash- > fn_zone[28]- > fz_next` trỏ tới `fn_hash- > fn_zone[27]`).
- `struct fib_node **fz_hash` - con trỏ tới một bảng hash table (của các nodes trong vùng).
- `int fz_nent` - số entry (node) trong vùng.
- `int fx_divisor` - số buckets trong bảng hash table đã kết hợp với vùng; có 16 buckets trong bảng cho hầu hết các vùng (trừ vùng đầu tiên- 0000 - phục vụ loopback device).
- `u32 fz_hashmask` - là mặt nạ (mask) ứng với bảng hash table của các entry; 15 (0x0F) cho hầu hêt các vùng, 0 cho vùng 0).
- `int fz_order` - số thứ tự của vùng con trong một vùng cha `fn_hash` (0 đến 32).
- `u32 fz_mask` - mặt nạ mạng của vùng (được định nghĩa =`((1<<(32-fz_order))-1`); ví dụ, phần tử đầu tiên (zero) được tính như sau: 1 dịch trái 32-0 lần (ta được 0x10000), sau đó trừ đi 1 (ta được 0xFFFF), và phần bù là (0x0000). Tính tương tự phần tử thứ 2 có netmask là 0x8000, các phần tử tiếp theo lần lượt là 0xC000, 0xE000, 0xF000, 0xF800, và phần tử cuối cùng (thứ 32) có netmask là 0xFFFF.

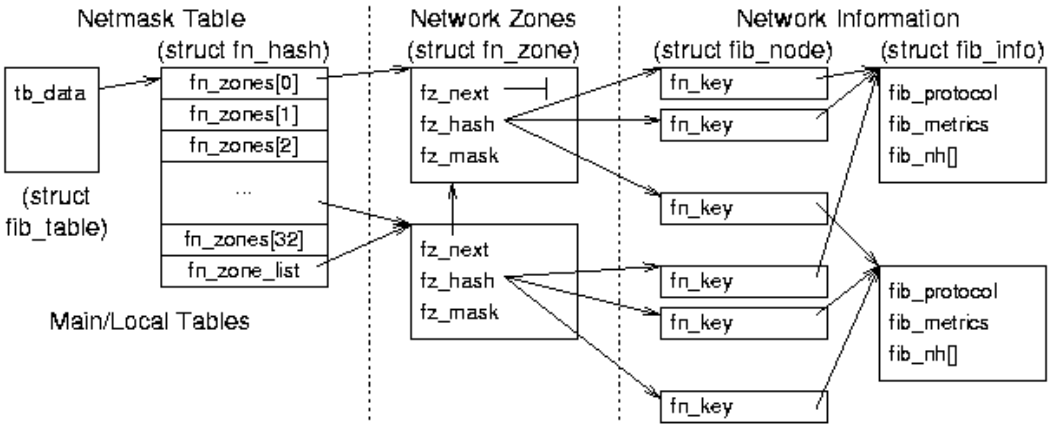
Cấu trúc Network Node Information **fib_node** - trong tệp *net/ipv4/fib_hash.c* - các cấu trúc này bao gồm thông tin duy nhất của từng tập hợp địa chỉ và một con trỏ tới một số thông tin (chẳng hạn như device và protocol); cấu trúc **Network Node Information** được xác định cho một network đã biết (là sự kết hợp duy nhất của source/destination/TOS).

- `struct fib_node *fn_next` - con trỏ tới node tiếp theo.
- `struct fib_info *fn_info` - con trỏ tới thông tin về node (nó dùng chung cho một số node khác).
- `fn_key_t fn_key` - hash table key - tối thiểu 8 bits cho địa chỉ đích (hoặc là 0 cho loopback device).

- Một số trường khác chứa thông tin về node ví dụ như `fn_tos` (type of service) và `fn_state`.

Cấu trúc **Network Protocol Information** (fib_info) - định nghĩa trong tệp *include/net/ip_fib.h* - các cấu trúc này chứa thông tin về protocol và hardware (của một giao diện xác định); một số mạng có thể đánh địa chỉ thông qua cùng một giao diện. Mỗi một cấu trúc fib_info có một giao diện như vậy.

- `fib_protocol` - số thứ tự của một giao thức mạng (ví dụ IP) được sử dụng định tuyến.
- `struct fib_nh fib_nh[0]` - chứa một con trỏ tới thiết bị sử dụng cho việc gửi và nhận truyền thông định tuyến.
- Một số trường khác bao gồm thông tin con trỏ tới danh sách, thống kê và dữ liệu tra cứu (như là `fib_refcnt` và `fib_flags`).



Hình 12: Forwarding Information Base (FIB) data relationships.

The Routing Cache

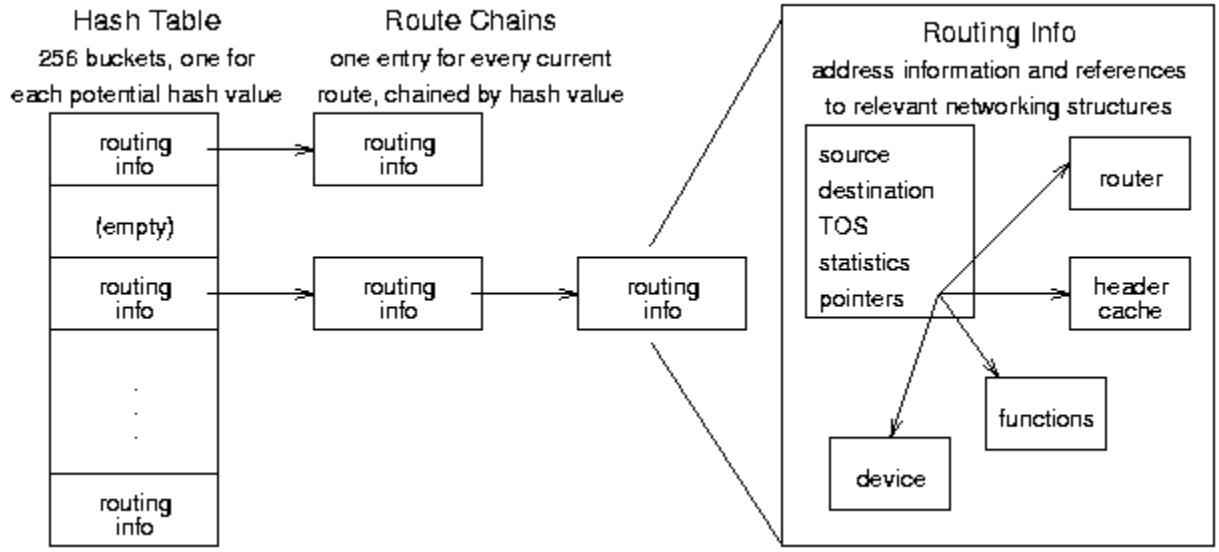


Figure 13: Routing Cache conceptual organization.

Routing cache là một phương pháp nhanh nhất để Linux tìm ra tuyến; nó lưu giữ tất cả các tuyến hiện tại đang sử dụng hoặc hiện tại đã được sử dụng trong một bảng **hash table**. Khi IP cần một tuyến, nó tìm đến một **hash bucket** tương ứng và tìm trong chuỗi các tuyến được lưu trữ trong đến khi tìm thấy, sau đó tiến hành gửi gói tin theo tuyến đó (xem phần trên). Các tuyến được móc (móc xích) vào nhau mọt trật tự (thường được sử dụng thì đặt lên trên) và có các bộ đếm thời gian và bộ đếm để có thể loại bỏ chúng ra khỏi bảng khi hết thời hạn sử dụng. Bạn có thể xem các hình 12, 13, 14 để thêm thông tin về các cấu trúc dữ liệu.

struct rtable *rt_hash_table[RT_HASH_DIVISOR] - đây là biến toàn cục chứa 256 buckets của móc xích trong các entry của **routing cache (rtable)**; hàm **hash** kết hợp với địa chỉ nguồn, địa chỉ đích và TOS để lấy một entry trỏ tới bảng (từ 0 đến 255). Nội dung của bảng này được liệt kê trong tệp */proc/net/rt_cache*.

Cấu trúc **Routing Table Entry** (**rtable**) - trong tệp *include/net/route.h* - cấu trúc này chứa các entry của **destination cache** và thông tin định danh của từng tuyến.

- **union < struct dst_entry dst; struct rtable* rt_next) > u** - đây là một entry trong bảng; cấu trúc **union** cho phép truy cập nhanh tới entry tiếp theo trong bảng bằng cách sử dụng trường **next** của **rtable** để trỏ tới entry của cache tiếp theo nếu được yêu cầu.
- **__u32 rt_dst** - địa chỉ đích.
- **__u32 rt_src** - địa chỉ nguồn.
- **rt_int iif** - giao diện của đầu vào.
- **__u32 rt_gateway** - địa chỉ của **neighbor** để chọn tuyến qua (phục vụ cho việc lấy một địa chỉ đích).
- **struct rt_key key** - cấu trúc này chứa khoá để tìm kiếm trong cache (**cache lookup key** - với các trường src, dst, iif, oif, tos, và scope).
- Một số trường khác chứa các cờ, kiểu, và một số thông tin khác.

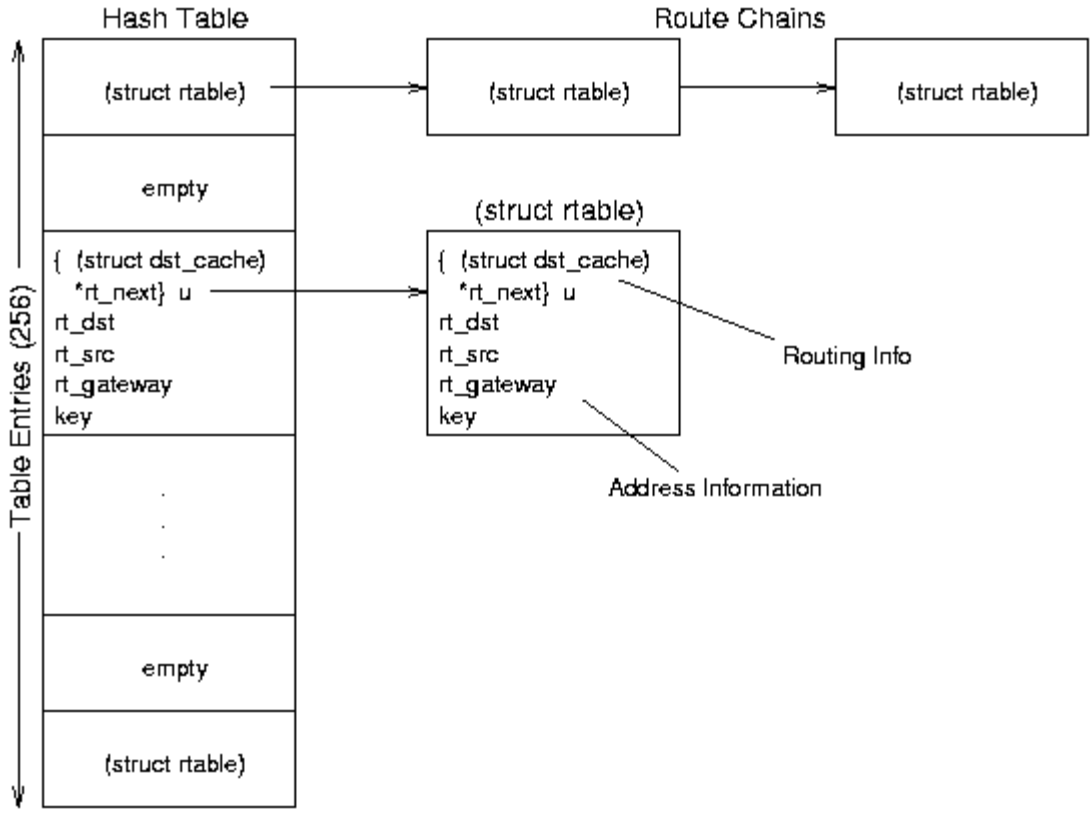
Cấu trúc **Destination Cache** (**dst_entry**) - trong tệp *include/net/dst.h* - cấu trúc chứa các con trỏ để xác định các hàm vào/ra và dữ liệu cho một tuyến.

- **struct device *dev** - thiết bị vào/ra của tuyến.
- **unsigned pmtu** - kích thước lớn nhất của gói tin của tuyến (**maximum packet size**).
- **struct neighbor *neighbor** - con trỏ tới **the neighbor** (liên kết tiếp theo) của tuyến.
- **struct hh_cache *hh** - con trỏ tới header của phần cứng (**hardware header cache**), nó tương tự cho các gói tin đi ra trên một liên kết vật lý, nó giúp cho việc truy cập rất nhanh và có thể sử dụng lại.

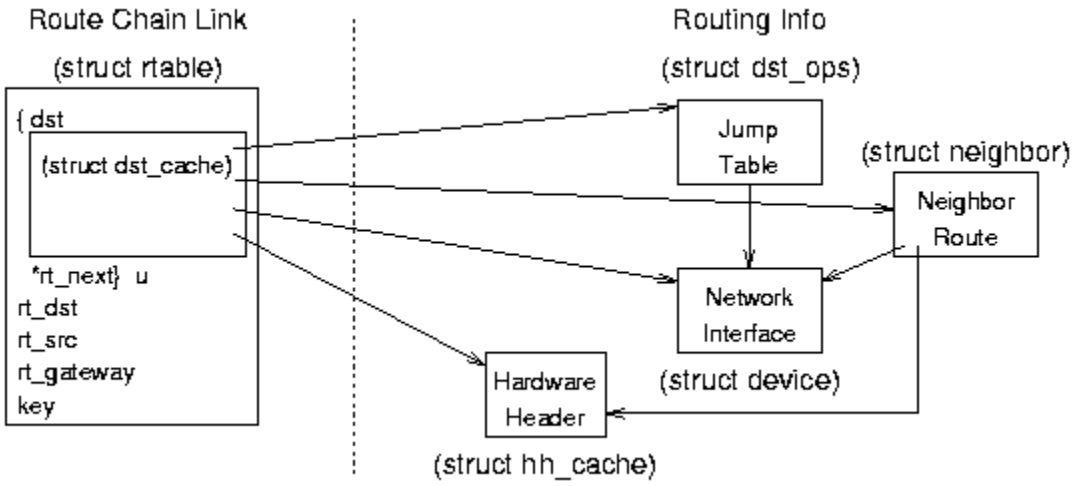
- **int (*input)(struct sk_buff*)** - con trỏ tới hàm vào của tuyến (đặc biệt là hàm **tcp_recv**()).
- **int (*output)(struct sk_buff*)** - con trỏ tới hàm ra của tuyến (đặc biệt là hàm **dev_queue_xmit()**).
- **struct dst_ops *ops** - con trỏ tới một cấu trúc chứa family, protocol, và check, reroute, và huỷ các hàm phục vụ cho tuyến này.
- Một số trường khác lưu giữ thông tin về thống kê và trạng thái và các liên kết tới các entry của bảng định tuyến khác.

Cấu trúc **Neighbor Link** (**neighbor**) - trong tệp *include/net/neighbor.h* - mỗi cấu trúc này đặc trưng cho một máy, chứa các con trỏ tới các hàm truy cập và thông tin của chúng.

- **struct device *dev** - con trỏ tới thiết bị mà được kết nối vật lý tới **neighbor** này.
- **struct hh_cache *hh** - con trỏ tới header mà luôn đứng trước đường gửi tới **neighbor** này.
- **int (*output)(struct sk_buff*)** - con trỏ tới hàm ra cho neighbor này (đặc biệt là hàm **dev_queue_xmit**()).
- **struct sk_buff_head arp_queue** - là thành phần đầu tiên trong hàng đợi của ARP cho đường liên quan đến **neighbor** này (vào/ra).
- **struct neigh_ops *ops** - con trỏ tới một cấu trúc chứa họ dữ liệu và các hàm ra cho liên kết này.
- Một số trường khác lưu giữ thông kê và trạng thái tham khảo cho các **neighbor** khác.



Hình 14: Routing Cache data structure relationships.



Hình 15: Destination Cache data structure relationships.

Cập nhật thông tin định tuyến

Linux chỉ cập nhật thông tin định tuyến khi cần thiết, nhưng các bảng này thay đổi theo các cách khác nhau. **Routing cache** thường hay bị thay đổi trong khi đó thì FIB thường không thay đổi (có thể thay đổi rất ít).

Bảng **neighbor table** thay đổi khi mà tuyến mạng bị thay đổic. Nếu một máy muốn gửi tới một địa chỉ trên một mạng cục bộ riêng (local subnet) nhưng nó lại không có trong bảng **neighbor table**, đơn giản là nó tung ra yêu cầu ARP và thêm một entry mới vào trong **neighbor table** khi nó nhận được trả lời. Kernel điêu khiển hầu hết những thayt đổi này một cách tự động.

FIB trên hầu hết các máy và thậm trí các routersn là không đổi; nó được điền vào khi khởi tạo với tất cả các vùng có thể để tìm đường kết nối tới tất cả các router và không thay đổi trừ khi một trong số các router đó donwn. Những thay đổi này thông qua lời gọi hàm **ioctl()** để thêm vào hoặc xoá các vùng.

Routing cache thay đổi thường phụ thuộc vào đường truyền thông. Nếu một máy muốn gửi các gói tin tới một địa chỉ ở xa, thì nó tìm địa chỉ trong **routing cache** (và **FIB** nếu cần) và gửi gói tin qua router tương ứng. Trên một máy đã được kết nối với một **LAN** nối một router vào Internet, thì tất cả các entry sẽ không những trỏ tới **neighbor** mà còn trỏ tới **router** (nhưng có thể có một số entry trỏ tới router). Tất cả được thực thi với các lời gọi hàm (mức IP) để tạo các tuyến và lập thời gian trong kernel để xoá chúng.

7.3 Các hàm trong Linux

Trong mục này chúng tôi đưa ra danh sách (theo alphabe) các hàm trong Linux được coi là quan trọng trong việc định tuyến và đồng thời đi phân tích cách thức làm việc của source code.

Functions/File	Descriptions
`arp_rcv()` có trong tệp `net/ipv4/arp.c`	-kiểm tra các lỗi (ví dụ: không có thiết bị ARP, không có thiết bị, gói đó không phải cho máy, không rõ kiểu thiết bị) -kiểm tra sự hoạt động: chỉ hiểu các thông báo REPLY và REQUEST -trích dữ liệu từ trong gói -kiểm tra các yêu cầu sai: các địa chỉ loopback hoặc multicast -kiểm tra khi nhận ra gói tin trùng địa chỉ (nếu cần) -nếu thông báo là một yêu cầu đúng và hàm `ip_route_input()` là đúng thì: + nếu gói tin từ một máy cục bộ thì: • gọi hàm `neigh_event_ns()` để tìm và cập nhật bảng neighbor đã gửi gói tin đó • kiểm tra thiết bị ẩn (không trả lời) • và gửi thông báo trả lời tới địa chỉ thiết bị đó + nếu gói tin không phải từ một máy cục bộ thì: • gọi hàm `neigh_event_ns()` để tìm và cập nhật bảng neighbor đã gửi gói tin đó • gọi hàm `neigh_release()`

	• nếu cần thì gọi hàm `arp_send()` với một địa chỉ • nếu không thì gọi hàm `pneigh_enqueue()` và trả về giá trị 0 -nếu là một thông báo trả lời thì: +gọi hàm `__neigh_lookup()` +kiểm tra xem nếu có nhiều trả lời ARP tới thì chỉ giữ lại ARP đầu tiên (nhanh nhất) +gọi hàm `neigh_update()` để cập nhật entry ARP và gọi hàm `neigh_release()` -giải phóng skbuffer và trả về giá trị 0.
`arp_send()` có trong tệp `net/ipv4/arp.c`	-kiểm tra để chắc chắn rằng thiết bị hỗ trợ ARP -cấp phát một skbuffer -điền thông tin vào header của buffer -điền thông tin ARP -gọi hàm `dev_queue_xmit()` để kết thúc gói.
`arp_req_get()` có trong tệp `net/ipv4/arp.c`	-gọi hàm `__neigh_lookup()` để tìm entry cho địa chỉ đã nhận được -copy dữ liệu từ entry của neighbor vào entry của arpreq -trả về 0 nếu tìm thấy hoặc trả về ENXIO nếu địa chỉ đó không trong bảng ARP.
`fib_get_procinfo()` có trong tệp `net/ipv4/fib_frontend.c`	-in các header và các kết quả của bảng `main_table`.
`fib_lookup()` có trong tệp `include/net/ip_fib.h`	-gọi hàm `tb_lookup() [= fn_hash_lookup()]` trên bảng `local_table` và `main_table` - nếu đã có một entry thì nó điền vào `fib_result` và trả về giá trị 0 -nếu không thì nó trả về giá trị lỗi tìm được.
`fib_node_get_info()` có trong tệp `net/ipv4/fib_semantics.c`	-hiển thị nội dung của `fib_node` và `fib_info` cho hệ thống file proc
`fib_validate_source()` có trong tệp `net/ipv4/fib_frontend.c`	-kiểm tra thiết bị và địa chỉ của gói vào -trả về mã lỗi (nếu có) -trả về 0 nếu gói tin là thích hợp.

`fn_hash()` có trong tệp `net/ipv4/fib_hash.c`	-thực hiện một hàm băm trên địa chỉ đích:
	`u32 h = ntohl(daddr)>>(32- fib_zone->fz_order);`
	`h ^= (h>>20);`
	`h ^= (h>>10);`
	`h ^= (h>>5);`
	`h &= FZ_HASHMASK(fz);//` với `FZ_HASHMASK=15` cho tất cả các vùng.

`fn_hash_get_info()` có trong tệp `net/ipv4/fib_hash.c`	-tìm các vùng trong bảng FIB sau đó gọi hàm `fib_node_get_info()` cho proc FS.
`fn_hash_lookup()` có trong tệp `net/ipv4/fib_hash.c`	-tìm các vùng trong bảng đã nhận được +tìm node trong từng bảng (giá trị khởi đầu là hash entry)

	entry) nếu đúng netmask (node và địa chỉ đích) thì: • kiểm tra TOS và trạng thái node • gọi hàm `fib_semantic_match()` để kiểm tra type của gói • điền dữ liệu vào `fib_result` và trả về giá trị 0. -trả về 1 nếu không tìm thấy gì
`fn_new_zone()` có trong tệp `net/ipv4/fib_hash.c`	-cấp phát bộ nhớ (trong kernel) cho vùng mới -cấp phát không gian nhớ đủ 16 node bucket cho vùng (trừ vùng đầu tiên 0.0.0.0 dành cho loopback) -lưu netmask (sử dụng n bit với n là vị trí của vùng trong bảng) -tìm vùng đã định trong bảng vùng cha -chèn vùng đó vào trong danh sách -cài vùng mới vào trong bảng vùng cha -cuối cùng giá trị trả về của hàm là vùng mới.
`fz_chain()` có trong tệp `net/ipv4/fib_hash.c`	-gọi hàm `fn_hash()` để lấy giá trị hash -trả về `fib_node` (trong `fib_zone` với thứ tự của hash).
`ip_dev_find()`có trong tệp `net/ipv4/fib_frontend.c`	-tìm kiếm và trả về thiết bị (với một địa chỉ trong bảng local table).
`ip_route_connect()` có trong tệp `include/net/route.h`	-gọi hàm `ip_route_output()` để nhận một địa chỉ đích -giá trị trả về nếu hàm trên thực hiện hoặc sinh ra một giá trị lỗi -nếu không thì nó xoá con trỏ tới tuyến và thực hiện lại.
`ip_route_input()` có trong tệp `net/ipv4/route.c`	-tính hash value cho địa chỉ đó -tìm trong bảng (giá trị khởi đầu là hash) giá trị kết nối (địa chỉ nguồn, đích, TOS, và IIF/OIF) -nếu tìm được thì cập nhật trạng thái và trả về routing entry -nếu không gọi hàm `ip_route_input_slow()`.
`ip_route_input_slow()` có trong tệp `net/ipv4/route.c`	-tạo một bảng routing table (lưu khoá) -kiểm tra các địa chỉ đặc biệt (loopback, broadcast, hoặc lỗi) -gọi hàm `fib_lookup()` để tìm tuyến -cấp phát bộ nhớ cho routing table entry mới -khởi tạo table entry với địa chỉ nguồn, đích, TOS, thiết bị ra và các cờ -gọi hàm `fib_validate_source()` để kiểm tra nguồn của gói -in ra thông báo và trả về lỗi nếu địa chỉ nguồn là sai -gọi hàm `rt_set_nexthop()` để tìm địa chỉ đích tiếp theo (neighbor) -trả về hàm `rt_intern_hash()` (cài tyến vào trong bảng routing table).
`ip_route_output()` có trong tệp	-tính hash value của địa chỉ đó -tìm giá trị kết nối (địa chỉ nguồn, đích, TOS và IIF/OIF)

net/ipv4/route.c	trong bảng với giá trị khởi đầu là hash -nếu tìm thấy một giá trị thì cập nhật trạng thái và trả về routing entry -nếu không thì gọi hàm ip_route_output_slow().
ip_route_output_slow() có trong tệp net/ipv4/route.c	-tạo một bảng routing table (lưu khoá) -nếu tìm được một địa chỉ nguồn thì gọi hàm ip_dev_find để xác định thiết bị ra -nếu không tìm được địa chỉ đích thì thiết lập loopback -gọi hàm fib_lookup() để tìm tuyến -cấp phát bộ nhớ cho routing table entry mới -khởi tạo table entry đó với địa chỉ nguồn, đích, TOS, thiết bị ra và các cờ -gọi hàm rt_set_nexthop() để tìm địa chỉ đích tiếp theo (neighbor - trả về hàm rt_intern_hash() (cài vào bảng routing table).
ip_rt_ioctl() có trong tệp net/ipv4/fib_frontend.c	-nhảy tới SIOCADDRT hoặc SIOCDELRT (nếu không thì trả về lỗi EINVAL) -kiểm tra quyền và copy tham số vào trong không gian kernel -chuyển đổi tham đã copy đó về cấu trúct rtentry - nếu xoá một tuyến, thì gọi hàm fib_get_table() và table->delete() -nếu không thì gọi hàm fib_new_table() và table->insert() –cuối cùng là giải phóng tham số đó và trả về giá trị 0 khi thành công.
neigh_event_ns() có trong tệp net/core/neighbour.c	-gọi hàm __neigh_lookup() để tìm địa chỉ trong bảng neighbor table -gọi hàm neigh_update() –trả về con trỏ tới neighbor đã chỉ ra.
neigh_update() có trong tệp net/core/neighbour.c	-kiểm tra các quyền để chỉnh sửa bảng -kiểm tra trạng thái neighbor nếu nó không là entry mới -so sánh địa chỉ đã nhận được: +nếu rỗng hoặc thiết bị không có địa chỉ thì sử dụng địa chỉ hiện tại +nếu khác nhau thì kiểm tra cờ; -sau đó gọi hàm neigh_sync() để kiểm tra xem neighbor vẫn còn up -cập nhật thời gian liên lạc với neighbor -nếu là entry cũ và entry mới (không thay đổi địa chỉ) thì trả về 0 -nếu địa chỉ mới khác với địa chỉ cũ thì thay đổi về địa chỉ mới -nếu địa chỉ mới và cũ trùng nhau thì trả về 0 -gọi hàm neigh_connect() hoặc neigh_suspect() để tạo hoặc kiểm tra kết nối -nếu trạng thái cũ là sai thì:

	+tìm các gói trong hàng đợi ARP, gọi hàm `output()` của neighbor với mỗi gói +lọc hàng đợi ARP -cuối cùng trả về 0.
`rt_cache_get_info()` có trong tệp `net/ipv4/route.c`	-in ra header và tất cả các thành phần của bảng `rt_hash_table` cho proc FS.

`rt_hash_code()` có trong tệp `net/ipv4/route.c`	-sử dụng địa chỉ nguồn, đích và TOS để xác định (và trả về) một giá trị hash:
	`hash = ((daddr&0xF0F0F0F0)>>4)|`
	`((daddr&0x0F0F0F0F)<<4);`
	`hash = hash^saddr^tos;`
	`hash = hash^(hash>>16);`
	`hash = (hash^(hash>>8)) & 0xFF;`

`rt_intern_hash()` có trong tệp `net/ipv4/route.c`	-đưa tuyến mới vào trong bảng định tuyến.

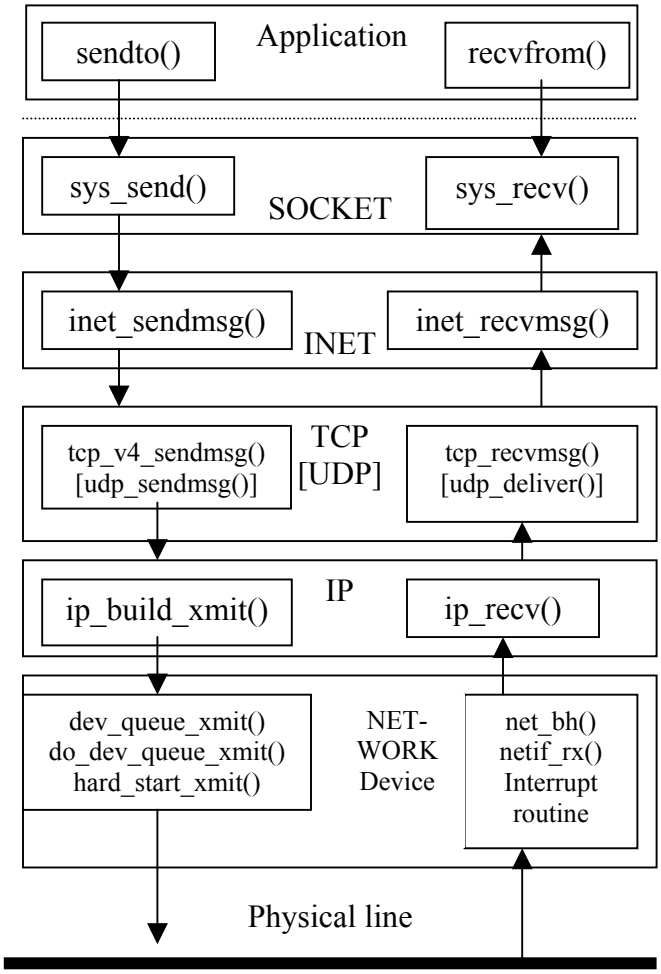
8.Kết luận

Qua các mục trên giúp cho chúng ta hiểu gần như toàn bộ hệ thống mạng trên Linux: từ cách khởi động hệ thống mạng đến việc gửi/nhận, routing và forwarding gói IP. Sâu hơn nữa là chúng ta đã tìm hiểu được nhiệm vụ của từng thủ tục liên quan đến toàn bộ hệ thống mạng. Chúng ta chỉ có một lưu ý là các thủ tục gửi/nhận dữ liệu của TCP và UDP qua giao thức TCP/IP của Linux là tương tự nhau (qua 5 tầng). Chỉ có một sự khác nhau đó là phương pháp sử lý của tầng TCP và UDP. Để hiểu rõ các bạn tham khảo hình minh hoạ dưới đây.

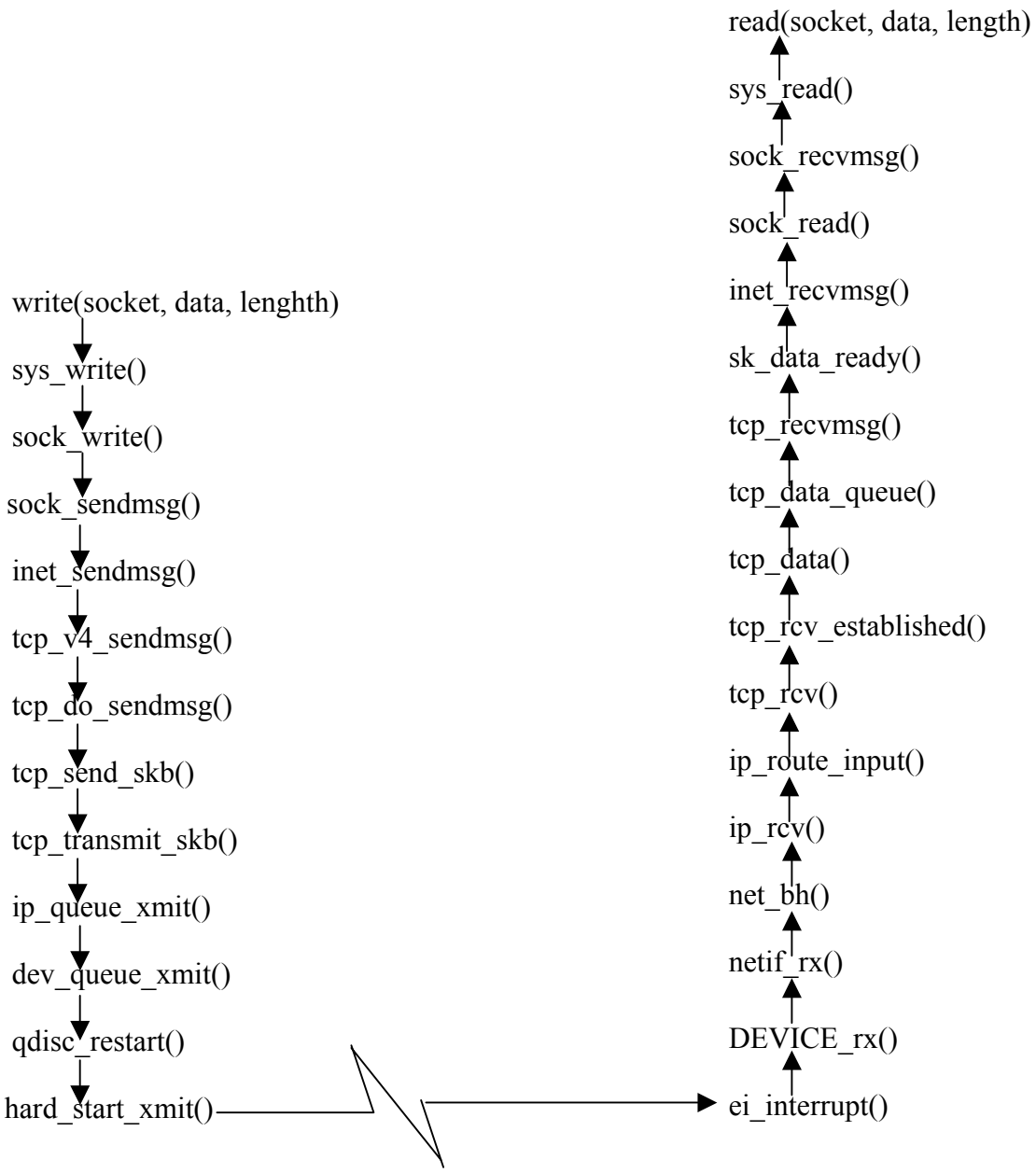
Để hiểu sâu về quản lý bộ nhớ cấu trúc dữ liệu sk_buff trong hệ thống mạng Linux, thì một điều hiển nhiên chúng ta phải hiểu được cách thức các gói tin đi qua chồng giao thức như thế nào? Như chúng ta đã biết, hệ thống mạng Linux được dựa trên nền tảng của giao diện BSD socket, nó hoạt động vào ra: **open()**, **read()**, **write()** và **close()** để thực hiện việc gửi các gói tin đến đích. Ví dụ, nếu có một socket đã được mở giữa 2 máy, thì máy đầu tiên có thể ghi vào socket máy kia có thể đọc từ socket đó. Ví dụ, khi chúng ta gọi:

write(socket, data, lenghth);

Với lời gọi này, nó sẽ gọi đến hàm **sys_write()**, là một phần của hệ thống file ảo của Linux (Virtual Files System). **sys_write()** chắc chắn rằng một hoạt động ghi có thể được thực thi trên fd (file descriptor) của socket, và có một ánh xạ vào bộ nhớ chứa dữ liệu đã được đọc. Khi đó fd này được dùng cho một socket, tiếp theo hàm sock_write() được gọi. **sock_write()** tìm kiếm cấu trúc dữ liệu socket đã kết hợp với inode của fd đó. Với mỗi một socket (thuộc **AF_INET family** - thường được dùng cho chuẩn TCP/IP networks), hàm **inet_sendmsg()** được gọi, nó thực hiện việc tách con trỏ socket sock (cấu trúc của INET socket), kiểm tra socket để chắc chắn rằng nó đang hoạt động, kiểm tra con trỏ tới giao thức, giá trị trả về của hàm là sk->prot[tcp/udp]- >sendmsg(). Việc nhận một gói tin của máy bên kia sử dụng Ethernet, khi Ethernet adapter nhận gói tin thì nó sinh ra một ngắt sẽ gọi hàm **ei_interrupt().** Hàm này sẽ gọi hàm **DEVICE_rx()** (phụ thuộc vào thiết bị), từ đó gọi hàm **netif_rx()**. Quá trình gửi/nhận gói tin qua chồng giao thức giữa 2 máy sẽ được tổng kết bằng hình dưới đây, để tìm hiểu đầy đủ chức năng mỗi hàm bạn có thể tham khảo trong các bảng trên.



Hình 16: Các thủ tục gửi/nhận gói tin của TCP [UDP]



Hình 17: Tổng kết các thủ tục gửi/nhận gói tin qua protocol stack.

Tài liệu tham khảo

1. Glenn Herrin, Linux IP Networking-A Guide to the Implementation and Modification of the Linux Protocol Stack
2. Alan Cox, Network buffer and memory management

Chương II
Lập trình mạng trong Linux

Hệ điều hành Linux áp dụng chuẩn công nghiệp Berkeley socket API, socket này có nguồn gốc trong sự phát triển BSD Unix (4.2/4.3/4.4 BSD). Trong chương này, chúng ta sẽ xem xét cách để quản lý bộ nhớ và bộ đệm đã được cài đặt trong tầng mạng và trong các trình điều khiển thiết bị của nhân Linux.

1. Các khái niệm chính

Tầng mạng được thiết kế theo hướng đối tượng (object-oriented), đây là một đặc điểm của nhân Linux. Cấu trúc chính của mã (code) mạng liên quan tới khởi tạo mạng và các ứng dụng socket tương ứng của Ross Biro và Orest Zborowski.

- ***Thiết bị (Device) hoặc giao diện (Interface)***: Một giao diện mạng là mã (code) chương trình để gửi và nhận các gói dữ liệu. Thường một giao diện được sử dụng cho một thiết bị vật lý như một card Ethernet; tuy nhiên một vài thiết bị chỉ là phần mềm, ví dụ như thiết bị loopback được sử dụng để gửi dữ liệu tới chính máy đó.
- ***Giao thức (protocol)***: Mỗi giao thức là một ngôn ngữ mạng khác nhau. Một vài giao thức tồn tại hoàn toàn do các nhà phát minh chọn để sử dụng cho các lược đồ mạng thích hợp, trong khi các giao thức khác được thiết kế cho các mục đích đặc biệt. Trong nhân Linux mỗi giao thức là một module riêng biệt của code mà cung cấp các dịch vụ cho tầng socket.
- ***Socket:*** Một socket là một kết nối trong mạng, nó cung cấp file I/O và tồn tại như là bộ mô tả file (file descriptor) đối với chương trình người dùng. Trong nhân Linux mỗi socket là một cặp cấu trúc biểu diễn giao diện socket mức cao và giao diện socket mức thấp.
- ***sk_buff:*** Tất cả các bộ đệm (buffer) được sử dụng trong tầng mạng là `sk_buffs`. Điều khiển cho những bộ đệm này được cung cấp bởi các thủ tục thư viện chính mức thấp, các thủ tục này có sẵn cho tất cả các hệ thống mạng. `sk_buffs` cung cấp bộ đệm chung và các tiện ích điều khiển dòng được cần bởi các giao thức mạng.

2. Cài đặt `sk_buffs`

Mục đích chính của các thủ tục sk_buffs là cung cấp một phương pháp quản lý bộ đệm nhất quán và hiệu quả cho tất cả các tầng mạng, và do tính nhất quán nên có thể cung cấp `sk_buff` mức cao hơn và các tiện ích quản lý socket tới tất cả các giao thức.

Một `sk_buff` là một cấu trúc điều khiển cùng với một khối bộ nhớ được gắn vào. Hai tập chính của các hàm này được cung cấp trong thư viện `sk_buff`. Tập đầu tiên bao gồm các thủ tục để thao tác với các dánh sách liên kết hai chiều của `sk_buffs`; Tập các hàm thứ hai để điều khiển bộ nhớ được gắn vào. Các bộ đệm

được lưu giữ trên một danh sách liên kết đã được tối ưu cho các thao tác mạng chung bằng cách thêm vào cuối và loại bỏ từ chỗ bắt đầu. Do có nhiều chức năng mạng xảy ra trong khoảng thời gian của một ngắt nên các thủ tục này được viết để sử dụng bộ nhớ atomic.

Chúng ta sử dụng danh sách các thao tác (operations) để quản lý nhóm các packet khi chúng đến từ một mạng và khi chúng ta gửi chúng đến tới các giao diện vật lý. Chúng ta sử dụng các thủ tục thao tác bộ nhớ để xử lý nội dung của các packet theo một cách chuẩn và có hiệu quả.

Ở mức cơ sở nhất, một danh sách các buffer được quản lý bởi các hàm như sau:

```
void append_frame(char *buf, int len)
{
  struct sk_buff *skb=alloc_skb(len, GFP_ATOMIC);
  if(skb==NULL)
    my_dropped++;
  else
  {
    skb_put(skb,len);
    memcpy(skb->data,data,len);
    skb_append(&my_list, skb);
  }
}

void process_queue(void)
{
  struct sk_buff *skb;
  while((skb=skb_dequeue(&my_list))!=NULL)
  {
    process_data(skb);
    kfree_skb(skb, FREE_READ);
  }
}
```

Hai đoạn mã lệnh đơn giản trên giải thích cơ chế nhận packet. Hàm `append_frame()` tương tự như mã mà được gọi từ một ngắt bởi trình điều khiển thiết bị khi nhận một packet, hàm `process_queue()` thì giống mã mà được gọi để đưa dữ liệu vào các giao thức. Nếu ta nhìn trong file `/net/core/dev.c` ở hàm `netif_rx()` và `net_bh()` thì ta sẽ thấy rằng hai hàm này quản lý bộ đệm hoàn toàn tương tự. Nó phức tạp hơn khi chúng phải đưa các packet tới đúng giao thức và quản lý điều khiển dòng, nhưng thao tác cơ bản thì giống nhau. Điều này chỉ đúng nếu ta xem các bộ đệm (buffer) đi từ mã giao thức tới một ứng dụng người dùng.

Ví dụ trên cũng chỉ ra việc sử dụng một trong các hàm điều khiển dữ liệu, hàm `skb_put()`. Hàm này được sử dụng để dự trữ không gian trong buffer cho dữ liệu mà chúng ta muốn chuyển xuống.

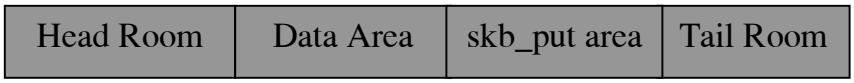
Ta hãy nhìn vào hàm `append_frame()`. Hàm `alloc_skb()` nhận được một buffer có độ dài `len` byte (hình 1), buffer này gồm:

- 0 byte ở đầu bộ đệm
- 0 byte dữ liệu
- len bytes ở Tailroom (tại nơi kết thúc của dữ liệu)

Hình 1. After alloc_skb

Hình 2. Sau khi gọi hàm skb_reserve

Hình 3. Một sk_buff chứa dữ liệu



Hình 4. Sau khi skb_put được gọi trên một buffer

Hình 5. Sau khi gọi skb_push trên buffer trước đó

Hàm `skb_put()` (hình 4) mở rộng vùng dữ liệu lên trên trong bộ nhớ thông qua không gian còn trống ở cuối của buffer, và bởi vậy đã dự trữ được không gian cho `memcpy()`. Nhiều thao tác (hoạt động) mạng gửi dữ liệu thêm vào không gian bắt đầu của frame mỗi lần thao tác gửi được thực thi, bởi vậy phần đầu có thể được thêm vào các packet đó. Vì lý do này, hàm `skb_push()` (hình 5) được gọi để thêm dữ liệu vào đầu của một buffer.

Ngay sau khi một buffer đã được cấp phát, tất cả room (chỗ) có sẵn thì dồn về cuối. Một hàm khác `skb_reserve()` (hình 2) có thể được gọi trước khi dữ liệu được thêm vào. Hàm này cho phép ta tạo ra một khoảng trống phía trước buffer. Do đó, nhiều thủ tục gửi bắt đầu với đoạn mã lệnh sau:

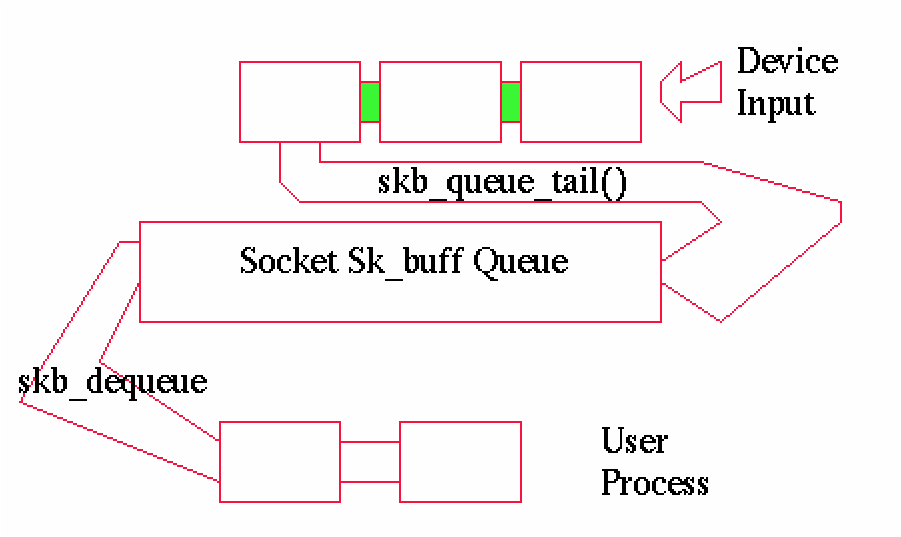
```
skb=alloc_skb(len+headspace, GFP_KERNEL);
skb_reserve(skb, headspace);
skb_put(skb,len);
memcpy_fromfs(skb->data,data,len);
pass_to_m_protocol(skb);
```

Trong các hệ thống như BSD Unix, bạn không cần biết bao nhiêu không gian mà ta sẽ cần, khi hệ thống này sử dụng dãy các buffer nhỏ (mbufs) cho các buffer mạng. Linux chọn sử dụng các buffer tuyến tính (linear) và nó đã "tiết kiệm" không gian (thường lãng phí một vài byte cho phép trường hợp xấu nhất), bởi vì buffer linear tạo ra nhiều thao tác khác nhanh hơn nhiều.

Linux cung cấp các hàm sau để thao tác với các danh sách:

- `skb_dequeue()` lấy buffer đầu tiên từ một danh sách. Nếu danh sách là rỗng thì hàm trả về một con trỏ NULL. Hàm này được sử dụng để lấy các buffer trong hàng đợi. Các buffer được thêm vào bằng các thủ tục `skb_queue_head()` và `skb_queue_tail()`.
- `skb_queue_head()` đặt một buffer ở đầu một danh sách.
- `skb_queue_tail()` đặt một buffer ở cuối của một danh sách, nó là một hàm hay được sử dụng. Hầu hết tất cả các hàng đợi (queues) được điều khiển với một tập các thủ tục xếp hàng dữ liệu bằng hàm này và một tập các hàm khác thiết lập việc xóa bỏ các mục (item) khỏi hàng đợi bằng `skb_dequeue()`.
- `skb_unlink()` xóa bỏ một buffer từ bất kỳ một danh sách nào chứa nó. Buffer đó thì không được giải phóng, chỉ đơn thuần là nó được xóa bỏ khỏi danh sách. Để tạo các thao tác dễ dàng hơn, ta không cần biết danh sách buffer chứa những gì, và ta có thể luôn luôn gọi skb_unlink() cho một buffer mà không ở trong bất kỳ một danh sách nào. Hàm này cho phép mã mạng xóa bỏ một buffer ra ngoài phạm vi sử dụng thậm trí khi giao thức mạng không biết rằng hiện tại ai đang sử dụng buffer đó. Một cơ chế khóa được cung cấp, bởi vậy một buffer mà đang được một trình điều khiểnthiết bị sử dụng thì nó không thể bị xóa.
- Một vài các giao thức phức tạp hơn như TCP giữ các frame theo trật tự và đảo trật tự đầu vào của nó khi dữ liệu được nhận. Hai hàm `skb_insert()` và `skb_apppend()` cho phép người dùng đặt một buffer trước hoặc sau một buffer chỉ ra trong một danh sách.
- Hàm `alloc_skb()` tạo ra một `sk_buff` mới và khởi tạo nó. Một buffer mới sẵn sàng để sử dụng nhưng giả định ta sẽ xác định vài trường trong đó để chỉ ra buffer này sẽ được giải phóng như thế nào. Bình thường vấn đề này được thực hiện bởi skb->fee = 1. Một buffer có thể được đặt cờ như là không thể giải phóng được bởi hàm `kfree_skb()` .
- Hàm `kfree_skb()` giải phóng một buffer nếu skb->sk được thiết lập. Hàm này giảm bộ nhớ sử dụng của tổng số socket(sk). Nó kích hoạt các thủ tục mức socket và giao thức (protocol-level) để tăng tổng này tránh giải phóng một socket với các bộ đệm còn tồn tại. Tổng số bộ nhớ là rất quan trọng khi các tầng mạng trong nhân cần biết bao nhiêu bộ nhớ được dùng cho mỗi kết nối, nhằm mục đích ngăn chặn các máy từ xa hoặc các tiến trình cục bộ khỏi việc sử dụng quá nhiều bộ nhớ.

- Hàm `skb_clone()` tạo một bản sao của một `sk_buff`, nhưng không sao chép vùng dữ liệu (vùng dữ liệu coi như là chỉ đọc).
- Thỉng thoảng một bản sao dữ liệu được cần để sửa đổi, hàm `skb_copy()` cung cấp một tiện ích tương tự như hàm `skb_clone()` nhưng hàm `skb_copy()` thì sao chép cả vùng dữ liệu.



Hình 6. Dòng các packet

3. Các thủ tục hỗ trợ mức cao hơn

Ý nghĩa của việc cấp phát và xếp hàng các buffer cho socket cũng liên quan đến các quy tắc điều khiển dòng và để gửi toàn bộ một danh sách tương tác cùng với các tín hiệu (signals) và tùy chọn khác như non blocking. Hai thủ tục sau được thiết kế để cho hầu hết các giao thức dễ thực hiện công việc này.

Hàm `sock_queue_rcv_skb()` được sử dụng để điều khiển dòng dữ liệu đến, bình thường nó được sử dụng trong dạng sau:

```
sk=my_find_socket(whatever);
if (sock_queue_rcv_skb(sk, skb) == -1)
{
      myproto_stats.dropped++;
      kfree_skb(skb,FREE_READ);
      return;
}
```

Hàm này sử dụng bộ đếm hàng đọc của socket nhằm ngăn chặn khối lượng lớn dữ liệu không xếp hàng tới một socket. Sau khi đến một giới hạn thì dữ liệu bị loại bỏ. Điều này đòi hỏi ứng dụng phải đọc nhanh dữ liệu, hoặc như là trong TCP, để giao thức thực hiện điều khiển dòng trên mạng. Thực ra,TCP yêu cầu máy gửi dừng lại (shut up) khi nó không thể xếp hàng dữ liệu.

Trên máy gửi, `sock_alloc_send_skb()` quản lý các tín hiệu (signal) điều khiển, cờ non-blocking và semantics of blocking cho đến khi có khoảng trống trong hàng gửi, cho nên bạn không thể liên kết tất cả bộ nhớ cùng với dữ liệu cho một giao diện chậm. Các hàm gửi của nhiều giao thức có hàm sau để kiểm tra xem liệu buffer đã được giải phóng hay chưa:

```
skb=sock_alloc_send_skb(sk,....)
if(skb==NULL)
    return -err;
skb->sk=sk;
skb_reserve(skb, headroom);
skb_put(skb,len);
memcpy(skb->data, data, len);
protocol_do_something(skb);
```

Đa số các hàm trong đoạn mã này chúng ta đã biết. Dòng quan trọng trong đoạn mã này là skb->sk=sk. Hàm `sock_alloc_send_skb()` thêm bộ nhớ cho buffer của socket. Bằng cách thiết lập trường skb->sk, chúng ta nói cho nhân biết rằng dù ai thực hiện hàm `kfree_skb()` trên buffer thì cũng cần thêm vùng nhớ cho buffer của socket. Bởi vậy, khi một thiết bị đã gửi một buffer và giải phóng nó, thì người dùng mới có thể gửi dữ liệu đến buffer đó.

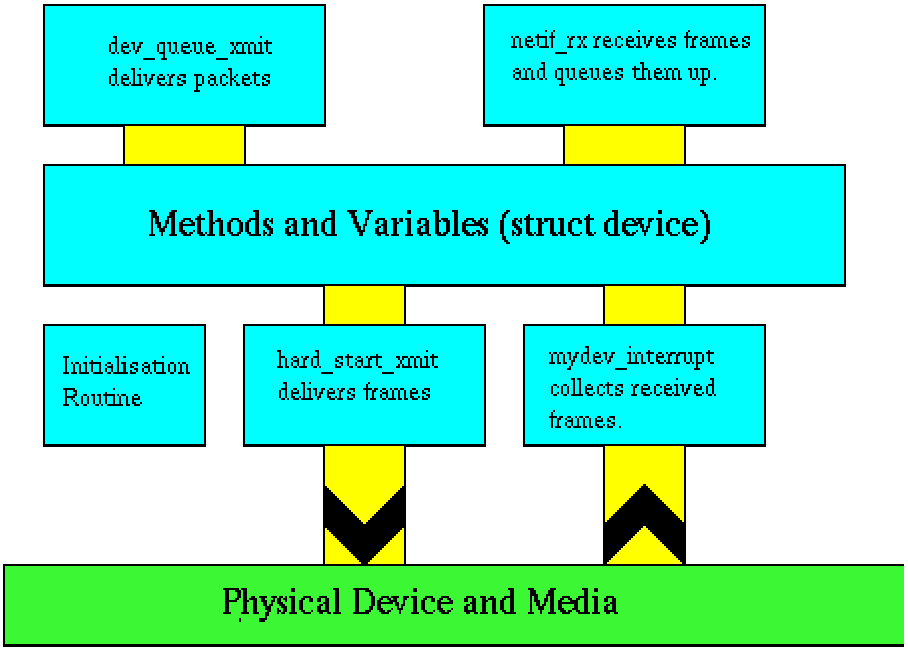
4. Thiết bị mạng

Tất cả các thiết bị mạng của Linux sinh ra từ cùng một giao diện, nhưng nhiều hàm có sẵn trong giao diện đó thì không cần cho tất cả các thiết bị. Hướng đối tượng được sử dụng, mỗi thiết bị là một đối tượng với một loạt các hàm (methods) tạo thành một cấu trúc. Mỗi hàm được gọi với thiết bị là đối số đầu tiên. File drivers/net/sketeton.c chứa bộ khung của một trình điều khiển thiết bị mạng.

4.1 Cấu trúc cơ bản của một thiết bị mạng

Mỗi thiết bị mạng xử lý việc truyền các buffer mạng từ các giao thức tới môi trường vật lý, và nhận, giải mã (decoding) các responses mà phần cứng sinh ra. Các frame đến được đưa vào các buffer, các buffer này được nhận diện bởi giao thức và được truyền tới neif_rx(). Sau đó hàm neif_rx() chuyển các frame tới tầng giao thức để xử lý thêm.

Mỗi một thiết bị cung cấp một tập các hàm thêm vào để quản lý vấn đề dừng (stopping), bắt đầu (starting), điều khiển (control) và bao bọc các packet. Tất cả các thông tin điều khiển được đặt cùng nhau trong cấu trúc thiết bị và chúng được sử dụng để quản lý mỗi thiết bị.



Hình 7. Cấu trúc của một thiết bị mạng

4.2 Đặt tên cho thiết bị mạng

Tất cả các thiết bị mạng trên Linux có một tên duy nhất mà không liên quan đến tên hệ thống file. Các thiết bị mạng bình thường không có một file trên hệ thống để biểu diễn nó, mặc dù ta có thể tạo một thiết bị mà gắn với các trình điều khiển thiết bị. Nhiều thiết bị cùng kiểu thì được đánh số tăng dần bắt đầu từ 0, ví dụ thiết bị Ethernet được đặt tên là "eth0", "eth1", "eth2"...Lược đồ đặt tên này có tầm quan trọng khi nó cho phép người dùng viết các chương trình hoặc cấu hình hệ thống trong thuật ngữ Ethernet card" mà không phải lo lắng về nhà sản xuất ra board và bắt buộc cấu hình lại nếu một board bị thay đổi.

Các tên sau đây được sử dụng cho các thiết bị chung:

- eth*n* Bộ điều khiển Ethernet, cho cả card 10 và 100Mbit/giây.
- tr*n* Các thiết bị Token ring
- sl*n* Thiết bị SLIP và AX.25 KISS mode
- ppp*n* Thiết bị PPP cả dị bộ (asynchronous) và đồng bộ (synchronous)
- plip*n* PLIP units; số phù hợp cho cổng máy in.
- tunl*n* Tunnel bao bọc IPIP
- nr*n* Thiết bị ảo NetROM
- isdn*n* Các giao diện ISDN được quản lý bởi isdn4linux
- dummy*n* Thiết bị Null
- lo Thiết bị loopback

4.3 Đăng ký một thiết bị

Một thiết bị được tạo bằng cách ghi vào một đối tượng `struct device` và chuyển nó tới hàm gọi `register_netdev(struct device *)`. Lời gọi này sẽ liên kết cấu trúc thiết bị vào bảng thiết bị mạng trong nhân. Khi nhân sử dụng cấu trúc này, ta không thể giải phóng nó đến tận khi ta unload thiết bị đó bằng cách gọi hàm `void unregister_netdev(truct device *)`. Hai hàm này bình thường được thực hiện ở thời gian khởi động hoặc khi load/unload module.

Nếu ta tạo nhiều thiết bị với cùng một tên thì thiết bị sẽ hỏng. Bởi vậy, nếu trình điều khiển thiết bị của ta là một module có thể nạp (loadable) thì ta nên gọi hàm `struct device *dev_get(const char *name)` để đảm bảo rằng tên đó chưa được sử dụng. Nếu tên thiết bị mà ta tạo đã được sử dụng rồi thì ta nên lấy tên khác nếu không trình điều khiển mới của ta sẽ lỗi. Khi ta phát hiện ra một xung đột, thì ta không nên sử dụng hàm `unregisster_netdev()` để loại bỏ thiết bị mà sử dụng tên đó.

Mã điển hình để đăng ký một thiết bị như sau:

```
int register_my_device(void)
{
  int i=0;
  for(i=0;i<100;i++)
  {
    sprintf(mydevice.name,"mydev%d",i);
    if(dev_get(mydevice.name)==NULL)
    {
      if(register_netdev(&mydevice)!=0)
        return -EIO;
      return 0;
    }
  }
  printk("100 mydevs loaded. Unable to load more.\n");
  return -ENFILE;
}
```

4.4 Cấu trúc thiết bị

Tất cả các thông tin và các hàm cho mỗi thiết bị mạng được giữ trong cấu trúc thiết bị. Để tạo một thiết bị ta cần cung cấp cấu trúc với phần lớn dữ liệu mà ta sẽ được thảo luận dưới đây. Phần này đề cập đến như thế nào để thiết lập một thiết bị.

Tên

Trường tên chứa một con trỏ (string pointer) trỏ đến một tên thiết bị, nó có dạng mà ta đã nói đến ở trên. Trường tên có thể là “ ” (bốn dấu cách), trong trường hợp này nhân sẽ tự động gán một tên eth*n* tới nó. Đặc điểm đặc biệt này không nên sử dụng. Sau Linux 2.0, một hàm `dev_make_name("eth")` được thêm vào để hỗ trợ mục đích này.

Các tham số giao diên bus

Khối các tham số tiếp theo được sử dụng để duy trì vị trí của một thiết bị bên trong một không gian địa chỉ cho thiết bị của kiến trúc. Trường `irq` chứa ngắt mà thiết bị đang sử dụng, bình thường ngắt này được thiết lập ở thời gian khởi động hoặc bởi các hàm khởi tạo. Nếu một ngắt không được sử dụng, không biết hoặc không được gán thì giá trị zero sẽ được sử dụng. Ngắt có thể được đặt bởi nhiều cácg. Tiện ích `auto-iqr` của nhân có thể được sử dụng để thăm dò ngắt cho thiết bị, hoặc ngắt có thể được thiết lập khi nạp module mạng. Các trình điều khiển mạng sử dụng một biến nguyên toàn cục gọi là `irq` cho ngắt bởi vậy người sử dụng có thể nạp module bằng lệnh kiểu như `insmod mydevice irq=5`. Cuối cùng, trường IRQ có thể được thiết lập động bằng cách sử dụng lệnh `ifconfig`, lệnh này sẽ được thảo luận sau.

Trường `base_addr` là địa chỉ không gian I/O cơ sở ở đó thiết bị trú ngụ. Nếu thiết bị không sử dụng I/O hoặc đang chạy trên một hệ thống không có khái niệm I/O thì trường này sẽ được thiết lập là 0. Khi địa chỉ này có thể được thiết lập bởi người dùng, thì biến toàn cục `io` sẽ chứa trường này. Địa chỉ I/O của giao diện cũng có thể được thiết lập bằng lệnh `ifconfig`.

Hai khoảng (dải) bộ nhớ phần cứng dùng chung (hardware-shared memory) được định nghĩa cho các trường hợp như bus ISA chia sẻ bộ nhớ của cards Ethernet. Đối với các mục đích hiện tại, các trường `rmem_start` và `rmem_end` không được dùng nữa và chúng sẽ được nạp là 0. Hai địa chỉ `mem_start` và `mem_end` được nạp là địa chỉ bắt đầu và kết thúc của khối nhớ mà thiết bị sử dụng. Nếu không có khối nhớ chia sẻ nào được dùng thì hai trường này nên chứa giá trị là 0. Các thiết bị mà cho phép người dùng thiết lập bộ nhớ cơ bản sử dụng một biến toàn cục được gọi là `mem` và sau đó chính thiết bị đó thiết lập địa chỉ `mem_end` thích hợp.

Biến `dma` chứa kênh DMA mà thiết bị sử dụng. Linux cho phép DMA (giống như ngắt) được thăm dò tự động. Nếu không sử dụng kênh DMA hoặc kênh DMA chưa được thiết lập thì boards phần cứng gán cho biến này giá trị là 0 và không gắn nó tới bộ nhớ còn trống. Nếu người dùng có thể thiết lập kênh DMA thì biến toàn cục `dma` được sử dụng.

Một điều quan trọng ta nhận thấy rằng các thông tin vật lý được cung cấp để điều khiển và cho người dùng xem (cũng như là các hàm bên trong trình điều khiển đó) và không đăng ký những vùng này để ngăn chặn chúng được sử dụng lại. Do đó, trình điều khiển thiết bị phải cấp phát và đăng ký I/O, DMA và ngắt mà nó muốn sử dụng.

Trường `if_port` chứa kiểu phương tiện vật lý (physical media type) cho các thiết bị đa phương tiện (multi-media) như board combo Ethernet.

Các biến tầng giao thức

Nhằm mục đích cho tầng giao thức thực hiện một cách khôn ngoan (sensible), thì

thiết bị phải cung cấp một tập các cờ và các biến mà chúng cũng được duy trì trong cấu trúc thiết bị.

Biến `mtu` chứa kích cỡ gói (payload) lớn nhất mà có thể gửi trên giao diện này. Kích cỡ packet lớn nhất không gồm bất kỳ các header nào của tầng dưới cùng (bottom layer) của chính thiết bị cung cấp. Số này được sử dụng bởi tầng giao thức (như IP) để lựa chọn kích cỡ packet phù hợp để gửi. Mỗi giao thức áp đặt quy định của chính nó. Một thiết bị không dùng một frame 576 byte hoặc lớn hơn cho IPX. IP cần ít nhất 72 byte và không thực hiện dưới 200 byte. Trường `mtu` kích hoạt tầng giao thức để quyết định có đồng vận hành (co-operate) với thiết bị của bạn không.

Biến `family` thì luôn được thiết lập là AF_INET và chỉ ra họ (dòng) giao thức mà thiết bị đang sử dụng. Linux thì cho phép một thiết bị sử dụng nhiều họ giao thức cùng một lúc.

Trường kiểu phần cứng `type` được lấy từ một bảng các kiểu phương tiện vật lý. Các giá trị trong bảng này được sử dụng bởi giao thức ARP (giao thức này được sử dụng bởi các phương tiện hỗ trợ ARP), các giá trị thêm vào được gán cho các tầng vật lý khác. Các giá trị mới được thêm mỗi khi cả nhân và **net-tool** cần (net-tool là một gói mà chứa các chương trình như ifconfig). Các trường này được định nghĩa trong Linux trước 2.0.5 như sau:

Theo RFC1700:

`ARPHRD_NETROM`	NET/ROM(tm) devices.
`ARPHRD_ETHER`	10 and 100Mbit/second ethernet.
`ARPHRD_EETHER`	Experimental Ethernet (not used).
`ARPHRD_AX25`	AX.25 level 2 interfaces.
`ARPHRD_PRONET`	PROnet token ring (not used).
`ARPHRD_CHAOS`	ChaosNET (not used).
`ARPHRD_IEE802`	802.2 networks notably token ring.
`ARPHRD_ARCNET`	ARCnet interfaces.
`ARPHRD_DLCI`	Frame Relay DLCI.

Được định nghĩa bởi Linux:

`ARPHRD_SLIP`	Serial Line IP protocol
`ARPHRD_CSLIP`	SLIP with VJ header compression
`ARPHRD_SLIP6`	6bit encoded SLIP
`ARPHRD_CSLIP6`	6bit encoded header compressed SLIP
`ARPHRD_ADAPT`	SLIP interface in adaptive mode
`ARPHRD_PPP`	PPP interfaces (async and sync)
`ARPHRD_TUNNEL`	IPIP tunnels
`ARPHRD_TUNNEL6`	IPv6 over IP tunnels

ARPHRD_FRAD	Frame Relay Access Device.
ARPHRD_SKIP	SKIP encryption tunnel.
ARPHRD_LOOPBACK	Loopback device.
ARPHRD_LOCALTLK	Localtalk apple networking device.
ARPHRD_METRICOM	Metricom Radio Network.

Những giao diện được đánh dấu không sử dụng (not used) được định nghĩa kiểu nhưng không được hỗ trợ trong gói **net-tools** hiện tại. Nhân linux cung cấp các thủ tục hỗ trợ chung cho các thiết bị sử dụng Ethernet và token ring.

Trường `pa_addr` được sử dụng để chứa địa chỉ IP khi giao diện được kích hoạt. Các giao diện đóng thì biến này cũng bị xóa. `pa_brdaddr` được sử dụng để chứa địa chỉ broadcast, `pa_dstaddr` là địa chỉ đích của một liên kết point-to-point, `pa_mask` là IP netmask của giao diện. Tất cả những trường này có thể được khởi tạo là 0. Trường `pa_alen` chứa độ dài của một địa chỉ (trong trường hợp địa chỉ IP), và nó nên được khởi tạo là 4.

Các biến tầng liên kết (link layer)

Trường `hard_header_len` là số byte mà thiết bị cần được chuyển đến đầu của một buffer mạng. Giá trị này không phải bằng số byte của header vật lý sẽ được thêm vào, mặc dù số này thường được sử dụng. Một thiết bị có thể sử dụng giá trị này để cung cấp cho chính nó với một scratch pad ở đầu của mỗi buffer.

Địa chỉ phương tiện truyền thông vật lý (nếu có) được để tương ứng trong `dev_addr` và `broadcast` và chúng là mảng byte. Địa chỉ mà nhỏ hơn kích thước của mảng thì được chứa ở đầu tính từ bên trái. Trường `addr_len` được sử dụng để chứa độ dài địa chỉ phần cứng. Nhiều phương tiện không có địa chỉ phần cứng, trong trường hợp này, trường này sẽ được thiết lập là 0. Một vài các phương tiện khác mà địa chỉ phải được thiết lập bởi chương trình người dùng. Công cụ `ifconfig` cho phép thiết lập một địa chỉ phần cứng của giao diện. Trong những trường hợp này thì địa chỉ đó không cần được khởi tạo, nhưng nên cẩn thận để không cho phép một thiết bị bắt đầu truyền thông trước khi một địa chỉ đã được thiết lập.

Các cờ (flag)

Một tập các cờ được sử dụng để duy trì các tính chất của giao diện. Các cờ này là:

- `IFF_UP` Giao diện được kích hoạt. Trong Linux, hai cờ IFF_RUNNING và IFF_UP được quản lý như là một cặp, và tồn tại như là hai mục vì lý do tương thích (compatibility). Khi một giao diện không được đánh dấu là IFF_UP, thì nó có thể bị xóa bỏ. Không giống như BSD, một giao diện mà chưa thiết lập IFF_UP thì sẽ không bao giờ nhận các gói.
- `IFF_BROADCAST` Giao diện có khả năng broadcast.
- `IFF_DEBUG` Chỉ ra gỡ lỗi. Hiện tại cờ này không được sử dụng

- IFF_LOOPBACK Cờ này được thiết lập khi giao diện là loopback (lo)
- IFF_POINTTOPOINT Giao diện là một liên kết point to point (như SLIP hoặc PPP). Bình thường, một liên kết point to point không có netmask hoặc broadcast, nhưng nếu yêu cầu thì nó có thể được cho phép.
- IFF_NOTRAILERS Cờ này không được sử dụng.
- IFF_RUNNING Xem IFF_UP
- IFF_NOARP Giao diện không thi hành lệnh truy vấn ARP.
- IFF_PROMISC Nếu có thể, giao diện sẽ nghe tất cả các gói trên mạng. Cờ này được sử dụng điển hình cho việc giám sát mạng, mặc dù nó cũng được sử dụng cho bridging. Một hoặc hai giao diện như giao diện AX.25 thì luôn luôn trong chế độ promiscuous
- IFF_ALLMULTI Nhận tất cả các gói multicast. Một giao diện mà không thể thực thi hoạt động này nhưng có thể nhận tất cả các gói, thì sẽ vào trong chế độ promiscuous khi được yêu cầu thực thi nhiệm vụ này.
- IFF_MULTICAST Chỉ ra giao diện mà hỗ trợ truyền thông multicast IP.

4.5 Hàng đợi

Các gói được xếp hàng cho một giao diện được sử dụng bởi mã nhân. Bên trong mỗi thiết bị, buffs[] là một mảng các gói được xếp hàng cho mỗi mức ưu tiên của nhân. Điều này được duy trì bởi mã của nhân, nhưng phải được khởi tạo bởi chính thiết bị đó khi khởi động. Mã khởi tạo được sử dụng là:

```
int ct=0;
While (ct<DEV_NUMBUFFS)
{
      skb_queue_head_init(&dev->buffs[ct]);
      ct++
}
```

Tất cả các trường khác nên được khởi tạo là 0.

Thiết bị phải lựa chọn độ dài hàng đợi mà nó cần bởi thiết lập trường dev->tx_queue_len tới một số lớn nhất của các frame mà nhân sẽ xếp hàng cho thiết bị đó. Điển hình, trường này được thiết lập khoảng 100 cho Ethernet và 10 cho đường truyền serial. Thiết bị có thể thay đổi động giá trị này.

5. Các hàm (methods) thiết bị mạng

Mỗi thiết bị mạng phải cung cấp một tập các hàm cho các thao tác cơ bản mức thấp. Ngoài ra thiết bị cũng cung cấp một tập các hàm hỗ trợ để giao tiếp giữa tầng giao thức và tầng liên kết.

5.1 Setup

Hàm init được gọi khi thiết bị được khởi tạo và được đăng ký với hệ thống, nhằm

mục đích thực hiện bất kỳ việc kiểm tra mức thấp và sự kiểm tra cần thiết. Hàm này sẽ trả về một mã lỗi khi thiết bị không hiện diện, các vùng (areas) không được đăng ký hoặc khi nó không thể xử lý. Nếu hàm `init` trả về một mã lỗi, thì hàm `register_netdev()` cũng trả lại một mã lỗi và thiết bị đó không được tạo.

5.2 Truyền frame

Tất cả các thiết bị phải cung cấp một hàm truyền. Một thiết bị có thể tồn tại mà không thể truyền. Trong trường hợp này, thiết bị cần một hàm truyền mà đơn giản là giải phóng buffer được chuyển tới nó. Thiết bị giả (dummy device) có chức năng này.

Hàm `dev->hard_start_xmit()` được gọi để cung cấp cho trình điều khiển thiết bị một con trỏ thiết bị và buffer mạng (một `sk_buff`) để truyền thông. Nếu thiết bị không chấp nhận buffer này, thì hàm trả về giá trị 1 và thiết lập `dev->tbusy` tới một giá trị khác 0. Hành động này sẽ xếp hàng buffer. Nếu tầng giao thức quyết định giải phóng buffer mà trình điều khiển thiết bị đã từ chối, thì buffer đó sẽ không được cung cấp trở lại thiết bị. Nếu thiết bị biết buffer không thể được truyền trong tương lai gần (ví dụ, bởi vì tắc nghẽn), thì nó có thể gọi hàm `dev_kfree_skb()` để kết xuất (dump) buffer và trả về giá trị 0 để chỉ ra rằng buffer đó đã được xử lý.

Khi có buffer trống nên được xử lý. Buffer đó đã chứa tất cả các header rồi, bao gồm header của tầng liên kết (link), thì nó cần được nạp vào trong phần cứng để truyền. Hơn nữa, nếu buffer bị khóa, điều đó có nghĩa là trình điều khiển thiết bị hoàn toàn sở hữu buffer đó đến tận khi trình điều khiển thiết bị "từ bỏ" (không cần) nó. Nội dung của một `sk_buff` có đặc tính là chỉ-đọc (read-only), ngoại trừ trường hợp các con trỏ trỏ đến trước/sau (next/previous) là free, bởi vậy ta có thể sử dụng danh sách `sk_buff` gốc để xây dựng dãy các buffer bên trong.

Khi buffer được nạp vào trong phần cứng, hoặc vào trong thiết bị điều khiển DMA, khi phần cứng cho biết việc truyền đã hoàn thành, thì trình điều khiển thiết bị phải giải phóng buffer bằng cách gọi hàm `dev_kfree_skb(skb, FREE_WRITE)`. Ngay sau khi thực hiện cuộc gọi này, sk_buff tự động biến mất, và trình điều khiển thiết bị sẽ không tham khảo nó nữa.

5.3 Frame Headers

Các giao thức mức cao cần gắn các header mức thấp tới mỗi frame trước khi xếp hàng nó để truyền. Rõ ràng giao thức không muốn biết làm thế nào để gắn các header mức thấp cho tất cả các kiểu frame. Bởi vậy, tầng giao thức đưa xuống thiết bị với một buffer có ít nhất `dev->hard_header_len` byte trống ở đầu bộ đệm. Sau đó nó kích hoạt thiết bị mạng để gọi hàm `skb_push()` và gắn header này vào packet bởi sử dụng hàm `dev->hard_header`. Các thiết bị mà không có header tầng liên kết (như SLIP), thì hàm này được chỉ ra là NULL.

Hàm `dev->hard_header` được gọi bởi cho nó các tham số: buffer có liên quan, con trỏ của thiết bị, nhận dạng giao thức (protocol identity), con trỏ trỏ tới địa chỉ nguồn và đích của phần cứng và độ dài packet (packet length). Do thủ tục có thể được gọi trước khi tầng thủ tục được hoàn toàn hình thành, cho nên nó dùng length parameter chứ không dùng buffer length.

Địa chỉ nguồn có thể là NULL có nghĩa là "thiết bị sử dụng địa chỉ mặc định", và địa chỉ đích có thể cũng là NULL có nghĩa là "không biết (unknown)". Nếu địa chỉ đích là unknown thì header đó chưa đầy đủ, một khoảng trống cần dành ra và một số bất kỳ byte có thể được thêm vào sẽ được điền vào. Hàm này cần phải trả về số ngược với số byte đã được thêm vào header (the function must then return the negative of yhe bytes of header added). Nếu header được xây dựng đầy đủ, thì hàm này trả lại số byte của header được thêm vào đầu của buffer đó.

Khi một header chưa đầy đủ, thì tầng giao thức sẽ cố gắng phân giải (resolve) địa chỉ cần thiết. Khi tình huống này xảy ra, hàm `dev->rebuild_header()` được gọi cùng với các tham số: địa chỉ mà header được đặt ở đó, thiết bị đang hỏi, địa chỉ IP đích và con trỏ buffer mạng. Nếu thiết bị không thể phân giải địa chỉ đó bởi các phương pháp có sẵn (bình thường sử dụng ARP), thì hàm sẽ ghi địa chỉ vật lý đó và trả về giá trị 1. Nếu header không thể được phân giải, thì hàm trả về giá trị 0 và buffer đó sẽ được xử lý lại ở lần sau, tầng thủ tục có lý do để tin răng ánh xạ địa chỉ là có thể.

5.4 Reception (nhận)

Không có hàm nhận trong một thiết bị mạng, bởi vì nó là một thiết bị mà được gọi để xử lý các sự kiện này. Với một thiết bị điển hình, ngắt (interrupt) thông báo cho trình xử lý rằng một packet đang sẵn sàng để nhận. Thiết bị gọi hàm `dev_alloc_skb()` để cấp phát một buffer có kích cỡ phù hợp, và đưa các byte lấy từ phần cứng vào buffer này. Tiếp theo, trình điều khiển thiết bị phân tích frame đó để xác định kiểu packet. Trình điều khiển đặt `skb->dev` cho thiết bị nhận frame. Nó thiết lập `skb->protocol` tới giao thức mà frame hiện diện, bởi vậy frame này được đưa đến đúng tầng giao thức. Con trỏ trỏ tới header tầng liên kết (link layer) được lưu giữ ở trong `skb->mac.raw`, và header tầng liên kết được xóa bỏ bởi hàm `skb_pull()`, do đó các giao thức không biết về nó. Cuối cùng, để cách ly giữa liên kết và giao thức, trình điều khiển thiết bị phải thiết lập `skb->pkt_type` tới một trong các giá trị sau:

- `PACKET_BROADCAST` broadcast ở tầng liên kết
- `PACKET_MULTICAST` multicast ở tầng liên kết
- `PACKET_SELF` Frame gửi tới máy này
- `PACKET_OTHERHOST` Frame gửi tới một máy khác

Kiểu cuối cùng thông thường được thông báo như kết quả của giao diện chạy ở promiscuous mode.

Cuối cùng, trình điều khiển thiết bị gọi hàm `netif_rx()` để chuyển buffer nên tầng giao thức. Các buffer này lại được xếp hàng để các giao thức mạng xử lý tiếp sau khi trình điều khiển ngắt trả về. Việc xử lý theo cách đó làm giảm đáng kể thời gian ngắt không cho phép làm gì và tăng tính đáp ứng. Một khi hàm `netif_rx()` được gọi, thì buffer đó không là "sở hữu" của trình điều khiển thiết bị nữa và nó không thể thay đổi hoặc tham khảo lại nữa.

Điều khiển dòng trên các packet nhận về được áp dụng ở hai mức bởi các thủ tục. Mức đầu tiên, một khối lượng dữ liệu lớn nhất còn để lại cho hàm `netif_rx()` xử lý. Mức thứ hai, mỗi socket trên hệ thống có một hàng đợi hạn chế khối lượng dữ liệu chưa được xử lý. Do đó, tất cả vấn đề điều khiển dòng được ứng dụng bởi các tầng giao thức. Bên truyền dữ liệu, biến `dev->tx_queue_len` được sử dụng để hạn chế độ dài hàng đợi. Bình thường, kích cỡ của hàng đợi là 100 frame, đây là hàng đợi có kích cỡ đủ lớn để gửi dữ liệu trên các liên kết nhanh. Trên các liên kết chậm như liên kết slip, thì hàng đợi được thiết lập là 10 frame.

Một công việc cần thực hiện khi nhận gói tin đối với hầu hết các thiết bị mạng hiện tại, là cần phải thực hiện cung cấp trước số byte cần thiết ở phần đầu của bộ đệm để dành chỗ cho phần IP header với độ dài tính theo word. Các trình điều khiển Ethernet thực hiện:

```
skb=dev_alloc_skb(length+2);
if (skb=NULL)
     return;
skb_reserve(skb, 2);
/* then 14 bytes of ethernet hardware header */
```

để đặt IP header vào trên vùng 16 byte, nó cũng chính là điểm bắt đầu của hàng đợi và giúp cho việc đưa ra các cải tiến về hiệu suất. Ở trên máy SPARC hoặc DEC Alpha những cải tiến này cũng rất rõ ràng.

6. Các hàm tùy chọn (optional functionality)

Mỗi thiết bị cung cấp thêm các hàm và các tiện ích tùy chọn cho các tầng giao thức. Không cài đặt các hàm này sẽ gây ra sự giảm sút các dịch vụ có sẵn thông qua giao diện, nhưng điều này sẽ không ngăn chặn hoạt động của mạng. Các hàm này được chia thành hai loại: configuration và activation/shutdown

6.1 Activation và Shutdown (Kích hoạt và tắt)

Khi một thiết bị được kích hoạt (cờ IFF_UP được thiết lập), thì hàm `dev->open()` được gọi. Gọi hàm này cho phép thiết bị thực hiện bất kỳ hành động nào (như cho phép giao diện, điều này cần trước khi giao diện đó được sử dụng). Hàm này trả về một lỗi gây ra cho thiết bị ở trạng thái down và khiến yêu cầu kích hoạt của người dùng phát sinh lỗi với một lỗi được trả lại bởi hàm `dev->open()`.

Hàm `dev->open` cũng có thể được sử dụng với bất kỳ thiết bị nào mà được nạp (load) dưới dạng module. Cần ngăn chặn thiết bị khỏi bị unloaded trong khi thiết bị đó mở; bởi vậy, macro `MOD_INC_USE_COUNT` phải được sử dụng bên trong hàm open này.

Hàm `dev->close()` được gọi khi thiết bị được cấu hình để sẵn sàng down và nên gỡ bỏ phần cứng theo một cách như thế để giảm thiểu công việc load của máy. Ngoài ra hàm này cũng được sử dụng để cho phép một module thiết bị được gỡ bỏ (unload) sau khi thiết bị đó down. Phần còn lại của nhân cũng được cấu trúc theo cách như thế khi một thiết bị được đóng, tất cả mọi tham chiếu đến thiết bị này bởi con trỏ thì bị xóa bỏ, nhằm mục đích đảm bảo rằng thiết bị được gỡ bỏ một cách an toàn từ một hệ thống đang chạy. Hàm close không được cho phép sinh lỗi.

6.2 Configuration và Statistics

Một tập các hàm cung cấp khả năng để truy vấn (hỏi) và để thiết lập các tham số vận hành. Một trong những hàm này đó là thủ tục `get_stats`, thủ tục này được gọi sẽ trả về một cấu trúc `enet_statistics` - khối các thông số cho giao diện đó. Khối này cho phép các chương trình người dùng như `ifconfig` thấy được việc nạp giao diện và các thông số của frame (frame truyền, frame nhận, frame lỗi...). Không cung cấp khối này có nghĩa là các thông số thống kê sẽ không có sẵn.

Hàm `dev->set_mac_address()` được gọi bất cứ khi nào một tiến trình siêu người dùng (superuser process) phát hành một ioctl của kiểu SIOCSIFHWADDR để thay đổi địa chỉ vật lý của thiết bị. Đối với nhiều thiết bị, hàm này không có ý nghĩa và đối với một số thiết khác thì hàm này không được hỗ trợ. Trong các trường hợp này, con trỏ hàm này được thiết lập là NULL. Một vài thiết bị có thể chỉ thực hiện thay đổi địa chỉ vật lý khi giao diện bị đóng (down). Đối với những thiết bị này, hàm kiểm tra cờ `IFF_UP`, và nếu nó được thiết lập thì hàm trả về –`EBUSY`.

Hàm `dev->set_config()` được gọi bởi hàm `SIOCSIFMAP` khi một người dùng thực thi một lệnh như `ifconfig eth0 irq 11`. Sau đó nó chuyển cấu trúc `ifmap` chứa I/O và các tham số giao diện khác. Hàm này thì không hữu ích cho đa số các giao diện .

Cuối cùng, `dev->do_ioctl()` được gọi mỗi khi một ioctl trong khoảng từ SIOCDEVIPRIVATE tới SIODEVPIRIVATE+15 được sử dụng trên giao diện. Tất cả các cuộc gọi ioctl này lấy một cấu trúc `ifreq`, cấu trúc này được sao chép vào trong nhân trước khi trình xử lý được gọi.

7. Multicasting

Một vài kiểu phương tiện vật lý nhất định (ví dụ như Ethernet) hỗ trợ frame

multicast ở tầng vật lý. Một frame multicast được nghe bởi một nhóm các máy (không phải tất cả) trên mạng, điều này tốt hơn nó đi từ một máy này tới máy khác.

Khả năng multicast của các card Ethernet là khác nhau. Hầu hết chúng rơi vào một trong 3 loại sau:

- Không lọc multicast. Card nhận tất cả frame multicast hoặc không nhận. Các card này có thể gây phiền toái trên mạng mà có nhiều truyền thông multicast, ví dụ như video hội nghị.
- Hash filter. Một bảng được tải vào card để đánh dấu những mục cho các multicast cần thiết. Phương pháp này lọc bỏ một số multicast không cần thiết nhưng không phải tất cả.
- Perfect filter. Phần lớn các card hỗ trợ đặc tính này cùng với một trong hai đặc tính vừa được nêu trên, bởi vì perfect filter chỉ giới hạn từ 8 đến 16 mục cho phép trong bảng.

Các giao diện Ethernet được lập trình để hỗ trợ multicast. Vài giao thức Ethernet (đặc biệt là Appletale và IP multicast) dựa vào Ethernet multicast. May thay, đa số các công việc liên quan tới multicast được thực hiện bởi nhân (xem net/core/dev_mcast.c).

Nhân hỗ trợ mã duy trì các danh sách địa chỉ vật lý mà giao diện sẽ cho phép multicast. Trình điều khiển thiết bị có thể trả về các frame trùng nhiều hơn danh sách multicast được yêu cầu nếu nó không thể thực hiện lọc hoàn hảo (perfect filter).

Bất cứ khi nào danh sách địa chỉ multicast thay đổi, trình điều khiển thiết bị gọi hàm `dev->set_multicast_list()`. Sau đó, trình điều khiển thiết bị có thể nạp lại bảng vật lý của nó. Mã thực hiện công việc này như sau:

```
if(dev->flags&IFF_PROMISC)
    SetToHearAllPackets();
else if(dev->flags&IFF_ALLMULTI)
    SetToHearAllMulticasts();
else
{
    if(dev->mc_count<16)
    {
        LoadAddressList(dev->mc_list);
        SetToHearList();
    }
    else
        SetToHearAllMulticasts();
}
```

Có một số card chỉ thực hiện unicast hoặc mode promiscuous. Trong những trường hợp này trình điều khiển thiết bị phải chuyển tới mode promiscuous khi có yêu cầu

multicast. Nếu điều này được thực hiện, trình điều khiển thiết bị phải tự nó thiết lập cờ `IFF_PROMISC` trong `dev->flags`.

8. Các thủ tục hỗ trợ Ethernet

Thông thường, Ethernet là giao diện vật lý thường được sử dụng. Nhân cung cấp một tập các thủ tục hỗ trợ Ethernet chung mà trình điều khiển có thể sử dụng.

`eth_header()` là điều khiển Ethernet chuẩn cho thủ tục `dev-hard_header`, nó có thể được sử dụng trong trình thiết bị Ethernet bấy kỳ. Kết hợp với `eth_rebuild_header()` nó cung cấp tất cả các tìm kiếm ARP cần thiết để đặt Ethernet header vào các gói IP.

Thủ tục eth_type_trans() để xử lý raw Ethernet packet. Nó phân tích các header và đặt các giá trị skb->pkt_type và skb_mac, đồng thời trả về giá trị skb->protocol. Thủ tục này thường được gọi bởi trình điều khiển ngắt nhận Ethernet để phân loại gói.

eth_copy_and_sum() là thủ tục cuối cùng hỗ trợ Ethernet, nó khá phức tạp, nhưng mang lại những tiến bộ đáng kể cho memory mapped card. Nó copy và kiểm tra dữ liệu từ card vào sk_buff cùng một lúc. Như vậy sẽ không cần tính checksum và tăng IP throughput.

PHẦN II
CÁC SẢN PHẨM BẢO MẬT GÓI IP

A. PHẦN MỀM TRANSCRYPT

Chương 1
Giải pháp Transcrypt

1. Tổng quan

1.1 Các tầng mạng và mã hoá

Phần mềm mà chúng tôi xây dựng làTranscrypt, nó dựa trên trình CIPE (Crypto IP Encapsulation) của Olaf Titz. Các công việc cần phải làm là khai thác làm chủ hoạt động của hệ thống và thay đổi phần mật mã (bao gồm thuật toán mã dữ liệu và toàn bộ phần trao đổi khóa).

Transcrypt là một chương trình thực hiện mã hoá tầng IP. Nó có thể được sử dụng để xây dựng các router cho VPN (Virtual Private Network) và các ứng dụng tương tự.

Có thể can thiệp Mật mã vào một vài nơi khác nhau trong một cấu trúc mạng sẵn có, tương ứng với các lớp giao thức khác nhau. Chẳng hạn:

- Ở mức mạng (Network level): có thể thực hiện mã hoá các gói tin trao đổi giữa các máy. Thành phần thực hiện mã hoá sẽ được đặt gần driver (trình điều khiển thiết bị), nơi thực hiện việc gửi và nhận các gói tin.
- Ở mức socket (socket level): thực hiện mã kết nối logic giữa các chương trình đang chạy trên các máy khác nhau (kết nối TCP; tầng Giao vận hoặc tầng Phiên (Session) trong mô hình OSI). Thành phần mã hoá sẽ ngăn chặn hoặc uỷ quyền kết nối. SSH và SSL được thực hiện theo cách này.
- Ở mức ứng dụng (application level): bản thân các chương trình ứng dụng chứa thành phần mã hoá riêng và tự mã hoá dữ liệu của chúng. Một ví dụ điển hình và khá nổi tiếng là PGP, phần mềm thực hiện mã hoá thư.

Việc mã hoá ở mức thấp như cách làm việc của Transcrypt có ưu điểm là nó làm cho công việc trở lên trong suốt, không hề làm thay đổi với các phần mềm ứng dụng. Trong trường hợp mã hoá các gói IP, ta có thể tích hợp trong các bộ định tuyến IP (router) - giống như những "hộp đen" chỉ định tuyến truyền thông giữa các máy - bản thân các máy tính không thấy được việc định tuyến này. Vì vậy một "bộ định tuyến mã hoá" cũng có vẻ giống như một bộ định tuyến không mã hoá, các máy tính và các ứng dụng khác không hề thấy sự khác biệt nào. Dựa theo cách sử dụng này (sử dụng như bộ định tuyến), khi ta đưa mật mã tác động ở các mức cao hơn là không thể thực hiện.

Mặt khác, việc mã hoá ở mức thấp có nhược điểm là nó không bảo vệ được những kẻ tấn công trên mức cao hơn. Ví dụ: các ứng dụng trojan, khai thác lỗi trên phần mềm hệ thống hoặc giả lập từ các thiết bị đầu cuối.

1.2 Định tuyến IP và mạng riêng ảo (Virtual Private Network)

Một mạng riêng ảo (VPN) là một mạng thuộc về một tổ chức, sử dụng một dải địa chỉ riêng, trùng với kiến trúc mạng có sẵn. "Đường hầm IP trong IP" (IP-in-IP tunneling) có thể xây dựng các mạng VPN (theo IP) ở trên các kênh truyền thông (dựa trên IP) khác, chẳng hạn như Internet. "Đường hầm đã mã hoá" chống lại được một số kiểu tấn công chủ động (tạo ra các thông điệp giả) và thụ động (trêu đùa) trên truyền thông mạng. Trên kênh truyền thông mạng chỉ nhìn thấy dữ liệu đã được mã hoá.

Tuỳ theo việc chọn giao thức, tất cả thông tin của các gói dữ liệu gốc có thể được mã hoá. Bao gồm cả dữ liệu thực (payload) và phần đầu của TCP/IP mà không để lại dấu vết về địa chỉ và các dịch vụ thực tế được sử dụng. Các tấn công bằng "phân tích truyền thông" - cố gắng lấy thông tin hữu ích từ những gói tin này, chẳng hạn như "ai đang liên lạc với ai" - sẽ không thể thực hiện được. Thậm trí có thể chống lại với những kỹ thuật tinh vi hơn nữa để "phá ngang" truyền thông, những kẻ tấn công có thể xen các gói tin giả (không mang thông tin hữu ích) vào mạng mà không thể phân biệt được với các gói dữ liệu thực.

Định tuyến IP với trường hợp VPN bao gồm cả định tuyến kênh truyền thông mạng, mà trong hầu hết các trường hợp chỉ là một thiết lập Internet chuẩn và định tuyến của VPN trùng với nó. Đây là cách dễ nhất khi dải địa chỉ truyền thông và VPN không trùng nhau. Thông thường VPN sử dụng vùng địa chỉ 10.0.0.0/8 đến 192.168.0.0/16, không là phần địa chỉ Internet và do đó không bao giờ xung đột với định tuyến Internet thực: bất kỳ địa chỉ nào trong vùng này phải là địa chỉ của mạng cục bộ trong tổ chức sử dụng nó.

Các chuẩn IPSEC định nghĩa một tập giao thức, có thể được dùng để tạo các mạng VPN mã hoá. Dù sao, IPSEC là một tập giao thức phức tạp với rất nhiều các tuỳ chọn, bổ sung của một tập giao thức đầy đủ mà vẫn ít được sử dụng và một số phần (chẳng hạn việc quản lý khoá) vẫn chưa được giải quyết một cách đầy đủ. Transcrypt tiếp cận một cách đơn giản hơn, và rất nhiều chức năng được tham số hoá (chẳng hạn như lựa chọn thuật toán mã hoá sử dụng) là lựa chọn cố định khi cài đặt. Điều này hạn chế tính phức tạp nhưng cho phép bổ sung đơn giản hơn (do đó có thể gỡ rối dễ dàng và hiệu quả...)**.**

1.3 Cách làm việc của Transcrypt

Transcrypt “bao bọc” các gói tin IP (đã được mã hoá) bởi các gói tin UDP và gửi chúng bằng kỹ thuật UDP thông thường. Đây là sự khác biệt với việc bao bọc IP trong IP. Lựa chọn UDP bởi các lý do chính sau:

- Theo cách này thì nhiều điểm cuối khác nhau có thể dễ dàng phân biệt bởi số cổng.
- Số giao thức IP cho phép một hình thức đăng ký.

- Điều khiển các gói tin UDP dễ dàng hơn việc sử dụng số giao thức IP độc lập, đặc biệt trong việc thiết lập bức tường lửa.
- Đặc biệt, các ứng dụng mức người dùng cũng có thể điều khiển được ở tầng UDP chẳng hạn như SOCKS5.

Một kết nối Transcrypt luôn thực hiện kết nối chính xác 2 điểm cuối với nhau. Trong nhiều cách, liên kết làm việc giống như một liên kết PPP dial-up. Hiện hành mỗi liên kết có một khoá bí mật riêng (có độ dài 512 bit) được thoả thuận trước ở cả 2 đầu (không có người thứ 3 biết). "Khoá liên kết" này (còn được gọi là "khoá tĩnh") được sử dụng thoả thuận "khóa động" - được thay đổi thường xuyên, sử dụng cho việc mã hoá dữ liệu thực.

Với Transcrypt, ta cũng có thể thực hiện thoả thuận khoá bằng kỹ thuật khoá công khai, tương tự như gói SSH. Kỹ thuật này không cần tới các khoá bí mật. Đó là công cụ dùng để trao đổi khoá KEX (viết tắt từ Key Exchange).

1.4 Các thành phần của TRANSCRYPT

Gói phần mềm Transcrypt gồm có một kernel module và một chương trình điều khiển (driver). Kernel module thực hiện điều khiển gói tin IP: gửi và nhận các gói, bao bọc và mã hoá. Nó bổ sung thêm một "thiết bị mạng", mà hầu như có thể điều khiển giống như bất kỳ một thiết bị mạng nào khác. Quá trình cấu hình và thay đổi khoá được thực hiện bởi chương trình mức người dùng "transcryptd".

'transcryptd" có cách xử lý giống như chương trình "pppd". Đặc biệt, việc mở và đóng thiết bị Transcrypt gắn với việc khởi động hoặc kết thúc tiến trình 'transcryptd" (mỗi tiến trình phục vụ cho một thiết bị),

Chương trình "kex" là phần bổ sung độc lập cho driver "transcryptd" - thực hiện việc trao đổi, quản lý khoá và các tham số khác.

1.5 Mã nguồn

Module bao gồm một driver đầu ra, một driver đầu vào, các thủ tục bao bọc và các hàm khác. Driver đầu ra là phiên bản "tunnel" mới từ bản phân phối Linux. (được trộn trong module IPIP trong nhân 2.2.). Trong Linux 2.0, gói dữ liệu được chuyển bằng thành phần "IP forwarding" của nhân. Điều này có nghĩa là: việc chuyển gói (forwarding) phải được sự "cho phép" trong nhân và các gói đã được mã hoá, chính là các gói UDP với địa chỉ nguồn/đích chính là các địa chỉ "me" và "peer" đưa ra, được kiểm tra lại khi qua forwarding của firewall. Các tham số cho thiết bị TRANSCRYPT được khai báo trong hàm transcrypt.h, nó là một cấu trúc như sau:

```
/* A TRANSCRYPT device's parameter block */

#define TRANSCRYPT_MAGIC  (htonl(0x43495045))
struct transcrypt {
```

```
    __u32               magic;
    struct NET_DEVICE   *dev;
     /* Set by user process */
    __u32               peeraddr;
    __u32               myaddr;
    __u16               peerport;
    __u16               myport;
    __u32               sockshost;
    __u16               socksport;
    short       cttl;
#ifdef Crypto_Blockcipher
    Key                 key, skey, rkey;
    #define key_e       key
    #define key_d       key
    #define skey_e      skey
    #define rkey_d      rkey
#endif
    unsigned long       tmo_keyxchg;
    unsigned long       tmo_keylife;
     /* Internal */
    unsigned long       timekx;
    unsigned long       timeskey;
    unsigned long       timerkey;
    int                 cntskey;
    int                 cntrkey;
    struct sock         *sock;
    int                 flags;
#ifdef LINUX_21
    char                recursion;
#endif
    pid_t               owner;
    /* Statistics */
#ifdef LINUX_21
    struct net_device_stats stat;
#else
    struct enet_statistics stat;
#endif
    /* Socket interface stuff */
    struct proto        *udp_prot;
    struct proto        transcrypt_proto;
};
```

Ta cũng có thể thấy được các nguyên hàm của các thủ tục, hàm được sử dụng chính cho thiết bị TRANSCRYPT trong file transcrypt.h này:

```
/* internal routines */
/* module.c */
extern void transcrypt_use_module(void);
```

```
extern void transcrypt_unuse_module(void);
extern int transcrypt_check_kernel(void);
/* device.c */
extern void transcrypt_prnpad(unsigned char *buf, int len) REGPARM;
extern void transcrypt_close(struct transcrypt *c);
extern const char *transcrypt_ntoa(__u32 addr) REGPARM;
/* sock.c */
extern int transcrypt_attach(struct NET_DEVICE *dev, struct siocsifcipatt *parm)
   REGPARM;
extern void transcrypt_fakenkey(struct transcrypt *c, char typ) REGPARM;
/* output.c */
#ifdef DEBUG
extern void transcrypt_dump_packet(char *title, struct sk_buff *skb, int dumpskb)
   REGPARM;
#endif
extern int transcrypt_xmit(struct sk_buff *skb, struct NET_DEVICE *dev);
/* encaps.c */
extern void transcrypt_encrypt(struct transcrypt *c, unsigned char *buf,
                               int *len, int typcode) REGPARM;
extern unsigned short transcrypt_decrypt(struct transcrypt *c, unsigned char *buf,
                                         int *len) REGPARM;
#ifndef VER_SHORT
extern void transcrypt_cryptpad(unsigned char *buf) REGPARM;
extern void transcrypt_cryptpad_iv(unsigned char *buf) REGPARM;
#endif
```

Driver đầu vào được mô phỏng theo hàm nhận UDP (hàm transcrypt_xmit) của nhân. Để kích hoạt nó, transcrypt phải thiết lập một socket vào chế độ đặc biệt với lệnh gọi hệ thống 'ioctl'. Điều này phải được thực hiện với socket UDP đã kết nối. Lệnh 'ioctl_attach' thay cho lệnh 'sendto' của socket, và 'recvfrom' thực hiện giải mã truyền thông bên trong và chỉ chuyển các khối trao đổi khoá cho lớp người dùng. Toàn bộ công việc giải mã và định tuyến lại truyền thông đến được thực hiện bên trong khối 'recvfrom'. Điều này có nghĩa rằng không giống như việc chuyển gói IP thông thường, nó có thể được gọi từ chế độ người dùng và thời gian CPU cần thiết để nạp cho tiến trình transcryptd, mặc dù dữ liệu không bao giờ được chuyển vào chế độ người dùng. 'sendto' mã khối giống như khối trao đổi khoá và gửi nó tới máy 'peer'. Socket không sử dụng 'read', 'write', 'select' hoặc chế độ phi khối nào.

Trước khi gắn socket, các tham số điều khiển thiết bị phải được thiết lập sử dụng lệnh gọi 'ioctl_setpar'. Thiết bị mạng chỉ có thể được mở nếu nó có một socket điều khiển. Khi socket điều khiển bị đóng, thiết bị mạng cũng đóng. Và ngược lại, khi đóng thiết bị mạng (bằng lệnh 'ifconfig') thì hệ thống cũng đóng luôn socket. Việc đóng lại sẽ xoá tất cả thông tin được thiết lập bởi transcryptd trên thiết bị.

2. Mô tả giao thức

Ý tưởng ban đầu của phần mềm này là cung cấp một phương pháp bảo mật (chống lại việc nghe trộm, bao gồm việc phân tích truyền thông, và tạo các thông điệp giả) trong truyền thông giữa các mạng con thông qua những mạng không bảo mật, chẳng hạn như Internet.

Giao thức này được thiết kế để có thể làm việc đơn giản và hiệu quả thông qua các kiến trúc mạng đã có sẵn, đặc biệt bằng việc bao bọc các gói tin UDP.

Giao thức này được chia thành hai phần: Mã hoá dữ liệu và trao đổi khoá động.

2.1 Mã hoá các gói

Mỗi gói tin IP được mã hoá toàn bộ, kể cả phần đầu của gói tin (header). Gói tin sẽ được thêm vào một số byte tùy từng độ dài gói tin cụ thể. Gói tin được đệm vào cuối cùng được thêm vào một octet chứa giá trị P (được mô tả ở dưới). Điều này đảm bảo rằng độ dài của gói tin luôn là bội của 16 octet.

Gói tin này sau đó được mã hoá bằng thuật toán mã khối MK1 (128 bit) trong chế độ CBC với một vector IV được mô tả ở dưới. Khoá sử dụng hoặc là khoá tĩnh hoặc khoá động được thực hiện thông qua việc trao đổi khoá công khai bằng công cụ KEX. Mã khối MK1 thực hiện mã hoá 128 bit dữ liệu bằng 512 bit khoá.

Gói tin đã mã hoá cũng chưa quyết định được với khối IV. Bit có thứ tự cao nhất của octet đầu tiên của IV là 0 nếu khoá sử dụng là khoá tĩnh, bằng 1 nếu sử dụng khoá động.

Giá trị P được đưa ra như sau: có 4 bít của P chỉ ra độ dài của phần thêm vào giữa các gói gốc và P. Các bit 2, 1 là mã kiểu, chỉ ra kiểu gói tin là gì.

Mã kiểu gồm có: 00 - dữ liệu, 01 - trao đổi khoá, 10 - được đặt trước, 11 - được đặt trước.

2.2 Trao đổi khoá

Khóa được trao đổi tự động bằng công cụ KEX, được chúng tôi mô tả chi tiết trong một mục khác.

2.3 Các vấn đề về bảo mật

Các gói tin thực tế được gửi đi không mang bất kỳ thông tin hữu ích khác có liên quan tới các gói tin IP gốc ban đầu cả về độ dài gói tin (được thêm vào thành là bội của 16). Điều này có thể bảo vệ một cách hiệu quả với hầu hết các khía cạnh trong việc phân tích truyền thông. Một thông tin khác là bit xác định để giải mã gói tin là bit báo là sử dụng khoá tĩnh hay khoá động. Thông thường khoá tĩnh chỉ được dùng trong những giai đoạn ngắn trong quá trình trao đổi khoá. Khoá động là khoá được thay đổi một cách thường xuyên. Những biện pháp phòng ngừa này là cần

thiết, bởi lẽ chương trình ứng dụng thám mã bằng các phương pháp tấn công các bản mã đã biết hoặc tấn công bản rõ chọn lọc từ việc chặn bắt được một khối lượng lớn các dữ liệu trên một khoá.

Tuy nhiên cũng có các gói tin IP mang một số các bản rõ dễ đoán hoặc biết trước trong những vị trí chắc chắn của chúng. Sự khai thác có hiệu quả thực tế này có thể làm giảm khả năng của khóa. Việc sử dụng số IV ngẫu nhiên cho mỗi gói tin có ý nghĩa trong việc chuyển lại các gói tin mã hóa thành các bản mã khác nhau.

3.Giải pháp can thiệp mật mã trong Transcrypt

3.1 Gói tin được gửi đi

Khi một gói tin cần chuyển đi từ mạng riêng này sang mạng riêng khác mà thuộc diện transcrypt bảo mật, thì trình transcrypt sẽ thực hiện việc bọc gói tin nguyên bản vào gói tin UDP (User Datagram Protocol – Gói tin người sử dụng).

Gói tin nguyên bản trước đó sẽ được mã hoá bằng việc gọi hàm:

```
int  length = skb->len;
transcrypt_encrypt(tunnel, skb->data, &length, TW_DATA);
```

Quá trình bọc gói tin nguyên bản bởi UDP header (phần đầu gói tin UDP) được thực hiện trong hàm transcrypt_xmit() trong file output.c:

Thực hiện việc cấp phát vùng nhớ cho UDP header:

```
/* Install our new headers */
 udph = skb->h.uh = (struct udphdr *) skb_push(skb, sizeof(struct
udphdr));
```

Gán địa chỉ UDP cho gói tin chuyển đi:

```
udph->source = tunnel->myport;
udph->dest   = tunnel->peerport;
dph->len    = htons(length+sizeof(struct udphdr));
```

Sau khi gán địa chỉ UDP cho gói tin thì trình sẽ thực hiện bọc gói tin UDP này bởi IP header được tạo thành tunnel giữa hai máy chạy transcrypt:

Địa chỉ đích là địa chỉ của máy bạn:

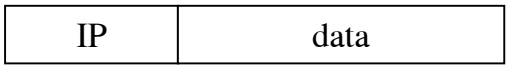
```
u32         dst = tunnel->peeraddr;

if (ip_route_output(&rt, dst, tunnel->sock->rcv_saddr, RT_TOS(tos),
                            tunnel->sock->bound_dev_if)) {
            dprintk(DEB_OUT, (KERN_NOTICE "%s: no route\n", dev->name));

      . . .
      goto tx_error_out;
}
```

Việc tạo địa chỉ IP header mới:

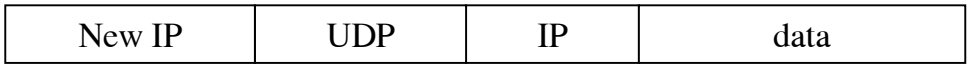
```
/*
 *    Push down and install the TRANSCRYPT/UDP header.
 */
iph->version    = 4;
iph->ihl        = sizeof(struct iphdr)>>2;
iph->tos        = tos;
iph->tot_len    = htons(skb->len);
ip_ident(iph, &rt->u.dst);
iph->frag_off   = df;
iph->ttl        = ttl;
iph->protocol   = IPPROTO_UDP;
iph->saddr      = rt->rt_src;
iph->daddr      = rt->rt_dst;

ip_send_check(iph);
```

Câu lệnh gán iph->protocol = IPPROTO_UDP chứng tỏ rằng gói tin mới tạo ra là gói tin UDP.

Chúng ta có thể mô tả cấu trúc gói tin trước và sau khi tạo tunnel:
Gói tin trước khi bọc:



Gói tin sau khi bọc:



3.2 Nhận gói tin

Ta nhận thấy việc xử lý gói tin nhận được hàm `struct sk_buff *transcrypt_decrypt_skb(struct transcrypt *c, struct sock *sk, struct sk_buff *skb, int *msgflag)` ở tệp sock.c thực hiện.

Khi nhận được gói tin do bên mã hoá gửi sang, thì bên nhận sẽ thực hiện công việc loại bỏ phần IP header và UDP header thêm vào.

Loại bỏ IP header thêm vào:

```
skb_put(n,skb->len);
memcpy(n->data, skb->h.raw, skb->len);
n->h.uh=(struct udphdr *)n->data;
```

Loại bỏ UDP header:

```
/* Pull out UDP header */
#ifdef LINUX_21
      n->nh.iph=(struct iphdr *) skb_pull(n, sizeof(struct udphdr));
```

```
#else
. . .
#endif
```

Sau khi loại bỏ IP header và UDP header thì trình sẽ thực hiện công việc giải mã gói tin để nhận được gói tin nguyên dạng ban đầu:

```
length=ntohs(skb->h.uh->len)-sizeof(struct udphdr);
switch (transcrypt_decrypt(c, n->data, &length)) {
...
}
```

Khi xử lý giải mã gói tin xong thì trình sẽ thực hiện gọi hàm netif_rx() để chuyển gói tin lên tầng mạng trên xử lý.

```
/* requeue */
netif_rx(n);
```

Chương 2
Phần mềm TRANSCRYPT

Gói phần mềm TRANSCRYPT được cung cấp dưới dạng file nén .tgz. Để thực hiện biên dịch, và cài đặt phần mềm này, ta tiến hành giải nén file này, chạy kịch bản configure để tạo ra các Makefile phục vụ cho quá trình biên dịch, cài đặt.

1. Quá trình cài đặt

1.1 Điều kiện tiên quyết

Để thực hiện biên dịch, cài đặt phần mềm TRANSCRYPT, hệ thống được cài đặt bản RedHat Linux 6.2 (nhân 2.2.14-5.0). Transcrypt được phát triển cho kiến trúc máy tính i386: các kiến trúc khác không làm việc được.

Trước tiên, phải bảo đảm rằng chúng ta có mã nguồn hoặc toàn bộ cây mã nguồn của nhân (kernel) đã được được cài đặt trên hệ thống (thường được đặt trong thư mục /usr/src/linux). Phiên bản và cấu hình của mã nguồn của nhân phải đúng với nhân mà nó sẽ chạy, nếu không tiến trình biên dịch module sẽ bị "lỗi". Tiếp theo, chúng ta cũng phải sử dụng cùng phiên bản của các trình biên dịch (compiler) đã biên dịch nhân. Sau khi cấu hình lại và biên dịch lại nhân, bạn cũng đừng quên biên dịch lại module Transcrypt (điều này cũng thường phải áp dụng với tất các module khác, chẳng hạn như các driver cho card mạng). Thông thường để đảm bảo module có thể làm việc với nhân.

Bảo đảm rằng bạn đã đặt chức năng "IP Forwarding", thông thường bằng lệnh:

```
echo 1 > /proc/sys/net/ipv4/ip_forward
```

trên những hệ thống khởi động từ nhân 2.2 hoặc mới hơn. Thiết bị 'urandom' cũng phải khả dụng trên hệ thống.

Một phiên bản các tiện ích về module (gói modutils - chứa các lệnh modprobe và insmod) cũng cần phải được cài đặt.

Công cụ KEX cần tới gói OpenSSL phiên bản 0.9.6 hoặc mới hơn. Nếu không có gói này, tất cả các phần còn lại của gói TRANSCRYPT vẫn được biên dịch và cài đặt một cách bình thường.

1.2 Mã nguồn của TRANSCRYPT

Sau khi giải nén gói chương trình TRANSCRYPT, bằng lệnh:

```
tar -xzf transcrypt-1.0.tgz
```

ta sẽ có thư mục transcypt-1.0 với các tệp sau đây:

TT	Thư mục /Tệp	Mô tả
1	***Thư mục gốc***	Thư mục ngoài cùng của gói phần mềm TRANSCRYPT.
1.1	configure.in	File chứa các tham số cấu hình cho file script

		configure.
1.2	configure	File shell script được dùng để thiết lập các tham số cấu hình, tạo ra các Makefile trước khi biên dịch.
2	***Thư mục conf/***	Thư mục này chứa các file cấu hình cần thiết cho các chương trình *autoconf, automake* sử dụng trong việc tạo ra các *Makefile* từ kịch bản configure ở trên.
2.1	aclocal.m4	
2.2	config.h.in	
2.3	Makefile-obj.in	
2.4	Makefile-top.in	
3	***Thư mục lib/***	Các hàm thư viện chung được sử dụng trong toàn bộ chương trình.
3.1	debug.c	Hàm logdebug.
3.2	dsprintf.c	Chứa hàm dsprintf.
3.3	getaddr.c	
3.4	gethex.c	
3.5	hex.c	
3.6	hexdump.c	
3.7	hexstr.c	
3.8	Makefile.h.in	File configure cần file này để tạo ra file Makefile cho thư mục này.
3.9	parseopt.c	
3.10	retstatus.c	
3.11	secchk.c	
3.12	setsig.c	
3.13	sighand.c	
3.14	socks_errlist.c	
3.15	socks5_cmd.c	
3.16	socks5_internal.c	
3.17	socks5_open.c	
3.18	transcrypt_syslog.c	Hàm transcrypt_syslog ghi các thông điệp để giúp cho việc gỡ rối chương trình, thông tin nhật ký.
3.19	transcryptlib.h	Hàm khai báo các prototype cho TRANSCRYPT.
3.20	xread.c	
3.21	xwite.c	
3.22	xwritev.c	
4	***thư mục kex/***	Thư mục chứa các file chương trình công cụ trao đổi khoá KEX (Key Exchange).
4.1	config.h	Chứa một số định nghĩa cần thiết cho chương

		trình KEX.
4.2	dhkey.h	Chứa một số định nghĩa các hằng sử dụng cho chương trình KEX.
4.3	lock.c	
4.4	main.c	Chương trình chứa hàm main.
4.5	Makefile	
4.6	mk1.h	Chứa một số khai báo về các hằng số cần cho chương trình mã hóa MK1.
4.7	negotiate.c	Chứa các hàm chính thực hiện việc trao đổi khóa.
4.8	p_sha1.c	Chương trình tạo ra dòng khoá từ thư viện của chương trình openssl.
4.9	packet.c	Các hàm và thủ tục điều khiển gói tin.
4.10	kex.h	Khai báo các hàm nguyên mẫu được sử dụng cho KEX.
4.11	proto.c	Chương trình chính điều khiển giao thức KEX.
4.12	***Thư mục kex/libgmp/***	Thư mục chứa các file thư viện cho tính toán số lớn, cần thiết để biên dịch chương trình KEX.
5	***Thư mục sample/***	Thư mục này chứa các file ip-down, ip-up và options mẫu, bạn có thể chỉnh sửa cho phù hợp với cấu hình mạng của bạn.
5.1	ip-down	script được gọi khi transcrypt tắt.
5.2	ip-up	script được gọi khi transcrypt khởi động.
5.3	options	File cấu hình cho TRANSCRYPT ở chế độ khởi động bằng khoá tĩnh.
6	***Thư mục transcrypt/***	Thư mục chứa các chương trình chính của phần mềm TRANSCRYPT.
6.1	blockcipher.o	File thư viện chứa chương trình thực hiện thuật toán mã khối MK1.
6.2	blockcipher.h	File khai báo các prototype cho chương trình blockcipher.
6.3	blockcipherconst.h	File khai báo một vài hằng được sử dụng cho mã khối MK1
6.4	cryptor.h	Khai báo một số hằng được sử dụng cho chương trình TRANSCRYPT.
6.5	device.c	Chứa các hàm khởi tạo, cấp phát, quản lý (đóng, mở) thiết bị cho transcrypt.
6.6	encaps.c	Thực hiện mã hoá gói tin, bao bọc gói tin theo định dạng gói tin UDP.
6.7	genoption.pl	Kịch bản viết bằng ngôn ngữ Perl để tạo ra 2 file option.h và option.c, cần thiết cho quá trình biên dịch. Hai file này được tạo dựa

		theo một số tuỳ chọn của nhân được biên dịch và file options.in.
6.8	ioctl.c	
6.9	ioctl.h	
6.10	module.c	Kiểm tra kiến trúc máy, kiểm tra nhân.
6.11	option.in	File genoptions.pl cần file này để tạo ra 2 file options.c và option.h.
6.12	output.c	Các hàm phục vụ cho phần gửi dữ liệu (dữ liệu ra) của TRANSCRYPT.
6.13	sock.c	Giao diện vào của TRANSCRYPT. Chứa các hàm xử lý gói tin khi nhận gói tin.
6.14	thruput.c	Đây là một chương trình độc lập, có nhiệm vụ giống như một "đồng hồ đo tốc độ", nó thực hiện đếm các byte dữ liệu gửi và nhận tới/từ một máy trong một lượng thời gian nào đó.
6.15	transcrypt.h	Chứa các định nghĩa về cấu trúc, các hằng, nguyên hàm chung cho tất cả các modules.
6.16	transcryptd.c	Đây là chương trình chính driver chế độ người dùng (ta còn gọi là chương trình daemon).
6.17	transcryptd.h	Chứa các hàm nguyên mẫu được sử dụng cho các thủ tục mức người dùng.
Tổng:	61 files	

1.3 Biên dịch

Sau khi bạn giải nén gói chương trình TRANSCRYPT (transcrypt.tar.gz) và được thư mục transcrypt-1.0 như mô tả ở trên. Bạn chạy file script ‘configure’ ở thư mục ngoài cùng của thư mục ‘transcrypt-1.0/’ để tạo ra các file cấu hình và các file Makefile phục vụ quá trình biên dịch.

Kịch bản cấu hình tự động (file 'configure' ở thư mục ngoài cùng) có nhiệm vụ tạo ra một số file header, các file Makefile cho từng hệ thống. Kịch bản này lấy các tham số sau từ dòng lệnh. Tất cả các tham số này đều có giá trị ngầm định đủ để cài đặt trên một hệ thống thông thường.

'--with-linux=dir':

Đường dẫn tới mã nguồn của nhân Linux (thông thường là /usr/src/linux/.

'--with-linux-include=dir':

Đường dẫn tới thư mục chứa các file tiêu đề, nếu bạn không có đầy đủ mã nguồn. Cây thư mục include này vẫn phải chứa tất cả các file được tạo bởi quá trình cấu hình nhân. Sử dụng toàn bộ cây mã nguồn này (cần thiết trên một số kiến trúc) vì trong trường hợp đó, các tham số 'Makefile' của nhân cũng được sử dụng.

'--with-ssl-includes=dir':

Đường dẫn tới thư viện OpenSSL, nếu script không tìm thấy nó.

'--enable-protocol=n':

Sử dụng giao thức bọc gói 'n'. Các giá trị được hỗ trợ trong TRANSCRYPT 1.0 hỗ trợ ngầm định là 3.

'--disable-debug':

Bỏ mã lệnh giúp gỡ rối chương trình. Tuỳ chọn này không thực sự hữu ích.

'--enable-logfacility=x':

Thiết lập chính sách nhật ký hệ thống cho 'transcryptd' và 'kex' (ngầm định và thường sử dụng là LOG_DAEMON).

'--enable-name=n':

Thiết lập tên hậu tố cho thư mục biên dịch.

'--disable-send-config':

Không gửi thông tin cấu hình sang máy kết nối. Thông tin này thông thường được gửi khi khởi động để giúp phân tích các lỗi, nhưng nó được gửi đi dưới dạng chưa được mã hoá (rõ).

'--disable-kex':

Không biên dịch và cài đặt công cụ KEX.

Kịch bản tìm kiếm các tham số chắc chắn trong các file tiêu đề của nhân, và nó tạo ra một thư mục mới với tên có dạng '2.2.14-i386-cb', mà quá trình biên dịch module và daemon sẽ được đưa vào đó. (Tên thư mục này viết tắt cho biên dịch cho nhân 2.2.14 trên máy i386, giao thức 3 -chữ c, sử dụng thuật toán mã hoá Blockcipher(b)). Nó tạo ra Makefile cho thư mục ngoài cùng dựa theo file conf/Makefile-top, và Makefile cho thư mục biên dịch đích từ file conf/Makefile-obj.

Đến đây, ta chỉ việc gõ một lệnh đơn giản ‘make’ để biên dịch tất cả mã nguồn. Hệ thống thực hiện biên dịch với từng thư mục; trước tiên là biên dịch trong thư mục lib/ để tạo ra file thư viện tĩnh libtranscrypt.a, sau đó tiến hành biên dịch trong thư mục đích (dạng ‘2.2.14-5.0-i386-cb’), tiếp theo nếu ta không dùng tuỳ chọn ‘--disable-kex’ thì nó sẽ thực hiện biên dịch luôn cả thư mục kex/:

```
all install clean::
     $(MAKE) -C lib $@
     $(MAKE) -C $(BUILD) $@
ifneq (,$(BUILD_KEX))
     $(MAKE) -C kex $@
endif
```

Trình biên dịch sẽ cảnh báo nếu có gặp phải một vấn đề gì khi biên dịch. Với những lỗi lớn ảnh hưởng tới quá trình biên dịch, chương trình có thể dừng quá trình biên dịch và đưa ra thông báo lỗi. Những lỗi này có thể do hệ thống có thể bị thiếu file tiêu đề, hay một file thư viện cần thiết nào đó.

- ***Biên dịch với nhiều nhân khác nhau***

Với cách sử dụng thư viện đối tượng riêng rẽ giúp ta có thể thực hiện biên dịch TRANSCRYPT cho các nhân khác nhau trong cùng một thư mục. Ví dụ: ta có một máy tính đang có thể chạy trên các nhân khác nhau. Trong trường hợp này ta có thể có các thư mục chứa mã nguồn của nhân dạng '/usr/src/linux-2.2.16', '/usr/src/linux-2.2.14-5.0', ... Ta thực hiện các lệnh sau:

configure --with-linux=/usr/src/linux-2.2.14-5.0
configure --with-linux=/usr/src/linux-2.2.16

Khi đó, chương trình sẽ tạo ra 2 thư mục là '2.2.16-i386-cb' và '2.2.14-5.0-i386-cb'. Và ta có thể chạy 'make' trong mỗi thư mục đối tượng một cách độc lập.

Một trường hợp khá phổ biến khác là có thể thiết lập một máy trung tâm để biên dịch các nhân cho các máy khác nhau. Ta có thể đổi tên các thư mục biên dịch của TRANSCRYPT với tuỳ chọn '--enable-name', có thể đặt tên này ở phần sau của tên máy đích. Giả sử ta cần biên dịch TRANSCRYPT cho máy bigbox với thư mục mã nguồn nhân là /usr/src/linux-2.2.14-bigbox, máy laptop với thư mục mã nguồn nhân là /usr/src/linux-2.2.16-small. Ta sẽ phải chạy kịch bản configure với các dòng lệnh như sau:

```
./configure --with-linux=/usr/src/linux-2.2.14-bigbox --enable-name=bigbox
make -C 2.2.14-i386-cb-bigbox
./configure --with-linux=/usr/src/linux-2.2.16-small --enable-name=laptop
make -C 2.2.16-i386-cb-laptop
```

Các nhà phân phối sản phẩm có thể chuẩn bị một tập các modules TRANSCRYPT khác nhau cho từng kiến trúc nhân khác nhau. Tuy nhiên cũng cần hết sức thận trọng khi biên dịch chéo như trên, hiện nay là không thể thực hiện được, bởi vì như chúng tôi đã trình bày ở phần 2.1 (Điều kiện tiên quyết) khi đó chúng ta cần phải biên dịch lại tất cả các module, driver cho nhân đó trên cùng hệ thống (với kiến trúc đó).

1.4 Cài đặt chương trình TRANSCRYPT

Thực hiện lệnh 'make install' với người dùng root để cài đặt các thành phần của phần mềm vào các vị trí cuối cùng của chúng. Các thành phần này bao gồm: một kernel module, có tên là transcrypt và số phiên bản của giao thức mà nó sử dụng và thuật toán mã hoá (transcryptcb). Các file Makefile sẽ sử dụng giá trị của các biến MODDIR, SBINDIR để xác định vị trí để cài đặt các thành phần trên. Trong đó, SBINDIR được chỉ vào /usr/sbin cho các chương trình daemon transcryptd-cb và công cụ trao đổi khoá kex; MODDIR chỉ ra thư mục chứa kernel module (thường có dạng /lib/modules/2.2.14-5.0/misc).

Nếu chương trình KEX đã được biên dịch, lệnh make install cũng sẽ cài đặt chương trình 'kex', thiết lập các thư mục cần thiết. Trong thư mục '/etc/transcrypt/', ta sẽ cần ít nhất là 2 file 'options' và 'ip-up'. Bạn cũng có thể sao

chép các file này từ thư mục 'sample' trong bản phân phối này, và hiệu chỉnh lại chúng cho phù hợp với những gì bạn cần.

Công cụ KEX và một số thành phần thư viện được tích hợp trong thư mục nguồn, chúng không phụ thuộc vào sự cấu hình cho từng cấu trúc máy cụ thể.

Các file sau khi cài đặt được thống kê trong bảng ở dưới.

TT	Tên file	Mô tả
1	Thư mục /etc/transcrypt	Thư mục chứa các file tuỳ chọn, cấu hình, file khoá công khai cho chương trình TRANSCRYPT.
1.1	options	File tuỳ chọn ngầm định được TRANSCRYPT sử dụng, chứa các tuỳ chọn về cấu hình mạng cho TRANSCRYPT. File này có chứa khoá bí mật được sử dụng trong liên kết bằng khoá tĩnh.
1.2	ip-up	Shell script được gọi khi chương trình daemon 'transcryptd' được khởi động. Bạn có thể copy file này từ thư mục sample trong gói cung cấp, và thiết lập các tham số, cấu hình phù hợp. File này thực hiện việc thêm định tuyến cho phù hợp với thiết lập của ta.
1.3	ip-down	Đây là shell script được gọi khi chương trình daemon 'transcryptd' bị tắt.
2	*Thư mục /etc/transcrypt/pk/*	Trong thư mục này chứa các file khoá công khai của các đầu kết nối khác.
3	*Thư mục /etc/transcrypt/run*	Được tạo để chứa các file khoá khi chạy công cụ trao đổi khoá động kex.
4	Thư mục /lib/modules/2.2.14-5.0/misc	
4.1	transcrypd-cb.o	Kernel module phục vụ cho thiết bị transcrypt.
5	*Thư mục /usr/sbin*	
5.1	transcryptd	Chương trình daemon thực hiện điều khiển thiết bị 'transcryptd'.
5.2	pktranscrypd	Công cụ thực hiện việc trao đổi khoá động.
5.3	file /var/log/transcrypt	Ghi thông tin nhật ký của trình transcrypt.

1.5 Thiết lập cấu hình

Có 2 cách để thực hiện ở bước này tùy thuộc vào cách sử dụng khoá - sử dụng khoá bí mật hay khoá tự động với công cụ trao đổi khoá KEX. Nếu sử dụng khoá bí mật ta chỉ cần cấu hình file tuỳ chọn /etc/transcrypt/options, khoá bí mật được ghi trong file này sau từ khoá 'key='. Tất nhiên ta cũng cần chú ý về vấn đề bảo mật với file dữ liệu này; cần phải đặt các quyền cho file này (chỉ người dùng root mới được

phép đọc hoặc ghi với file này), và có thể thiết lập thuộc tính không thể xoá đối với file này.

Phương pháp và cách thiết lập cấu hình cho công cụ trao đổi khoá tự động KEX đã được chúng tôi trình bày trong chương 4 (Trao đổi khoá).

Chú ý: với trao đổi khoá tự động bằng công cụ KEX, ta cần tiến hành nạp module transcryptcb.o trước khi gọi chương trình kex. Vì sau khi thực hiện trao đổi khoá xong chương trình KEX gọi chương trình daemon 'transcryptd-cb' với file tuỳ chọn mới.

1.6 Chạy chương trình TRANSCRYPT

Sau khi được cài đặt, phần mềm TRANSCRYPT được chạy bằng cách nạp module và chạydaemon ‘transcrypt’.

a. Tên chương trình

Tên module là ‘transrypt’, tiếp theo là phiên bản giao thức với 1 ký tự và ký tự đầu tiên của thuật toán mã hoá (ở đây là ‘c’ là phiên bản giao thức 3, và ‘b’ cho thuật toán mã hoá BlockCipher). Tên của thiết bị mà module quản lý là 'cryptcb' và tiếp đó là một số, ví dụ: 'cryptcb0'.

Chương trình daemon có tên là 'transcryptd-cb', 'c' là số phiên bản của giao thức được dùng và ký tự 'b' viết tắt về mã pháp được sử dụng (thuật toán Blockcipher MK1). Cụ thể trong gói phần mềm TRANSCRYPT này tên chương trình daemon này là 'transcryptd-cb', mà thông thường để cho gọn ta chỉ nói tới daemon 'transcryptd'.

Các tham số cấu hình của kernel module và daemon phải phù hợp với nhau (module sẽ kiểm tra điều này), tuy nhiên daemon không phụ thuộc vào phiên bản nhân.

b. Nạp các module

Kernel module được nạp vào nhân bằng lệnh sau:

```
modprobe modulename parameter=value ...
```

Module TRANSCRYPT chấp nhận các tham số bổ sung sau:

'transcrypt_debug=(number)'

Thiết lập mức gỡ rối. File 'transcrypt.h' định nghĩa các mức gỡ rối khác nhau, đặt là 0 nếu bạn không cần gỡ rối đầu ra. Gỡ rối đầu ra là việc đưa ra các thông điệp từ nhân, có ý nghĩa cho việc giải quyết các sự cố (biết được lỗi xảy ra từ đâu hay lý do xảy ra sự cố).

'transcrypt_maxdev=(number)':

Thiết lập số kênh mà module này quản lý. Ví dụ: với 'transcrypt_maxdev=4' các thiết bị 'cipcb0' đến 'cipcb3' là khả dụng. Giá trị lớn nhất cho phép với tham số này là 99. Thông thường, ta không cần phải thiết lập tuỳ chọn này, mà các kênh được cấp phát một cách tự động.

c. Chạy chương trình daemon 'transcryptd'

Daemon 'transcrypd' phải được chạy với người dùng root. Chương trình sẽ lấy các tham số từ tham số tuỳ chọn '-o file' (trên dòng lệnh gọi nó), tham số này chỉ ra file tuỳ chọn chứa các tham số mà bạn có thể đặt ở một nơi riêng nào đó (ngầm định nó sẽ tìm file này trong thư mục /etc/transcrypt/options).

Ngoại trừ chế độ gỡ rối, daemon tự chuyển xuống tiến trình nền và sử dụng lệnh 'syslog' để ghi lại các thông điệp nhật ký. Các thông điệp được ghi thông thường là các lỗi, các cảnh báo khi daemon bị ngắt.

Khi ta đóng thiết bị TRANSCRYPT (bằng lệnh ifconfig), thì tiến trình 'transcryptd' sẽ bị ngắt, và ngược lại khi ngắt daemon 'transcryptd' thì nó cũng sẽ đóng thiết bị TRANSCRYPT lại. Khi một thiết bị được đóng lại, các tham số cấu hình (gồm cả tất cả các khoá và các biến đếm) đều được xoá bỏ.

Khi thiết bị được 'dựng lên' (sau khi đã nạp module transcryptcb.o và chạy chương trình daemon), 'transcryptd' gọi chương trình '/etc/transcrypt/ip-up'. Các tham số cho file này được chúng tôi mô tả mẫu trong thư mục sample (trong bản phân phối). Nó sẽ đợi sau khi kịch bản này thi hành xong, dữ liệu mới có thể gửi qua thiết bị và trước khi nó được chuyển xuống tiến trình nền. Kịch bản này được gọi với đầu vào/ra chuẩn và lỗi được chuyển vào thiết bị '/dev/null'. Kịch bản này có nhiệm vụ thiết lập các định tuyến và các biến môi trường khác.

Tương tự như vậy, khi thiết bị TRANSCRYPT bị đóng lại, kịch bản '/etc/transcrypt/ip-down' được gọi. Bản thân chương trình 'transcryptd' sẽ ghi lại thông tin khi giao diện được đóng.

Chương trình 'transcrypt' sẽ ngắt khi có một lỗi nào đó xảy ra. Các lỗi thông thường có thể gặp là thông điệp "connection refused" từ máy bên kia, điều này có thể giúp ta phán đoán rằng máy bên kia không làm việc. Dữ liệu sẽ chỉ được gửi qua liên kết khi và chỉ khi cả hai đầu được "dựng lên" và đang chạy. Trong trường hợp một bên được "up" lên trước, khi có dữ liệu truyền qua lập tức thiết bị TRANSCRYPT ở đầu này sẽ bị đóng lại ngay lập tức.

2. Cấu hình phần mềm TRANSCRYPT

Trong phần này chúng tôi sẽ chỉ trình bày về cấu hình phần mềm TRANSCRYPT với việc thiết lập bằng khoá bí mật. Với thiết lập cấu hình phần mềm bằng khoá công khai được trình bày trong chương 4 (Trao đổi khoá). Cấu hình phần mềm TRANSCRYPT bằng khoá bí mật cần tới 3 file sau: /etc/transcrypt/options,

/etc/transcrypt/ip-up và /etc/transcrypt/ip-down. Trong đó 2 file ip-up và ipdown là 2 file shell script, có thể copy từ thư mục sample/ trong bản phân phối TRANSCRYPT, và chỉnh sửa lại cho phù hợp với cấu hình mạng của bạn. Trong đó quan trọng là file ip-up cần phải được sửa dòng sau:

route add -net 129.3.0.0 dev crypt0 gw 129.3.0.0

Cần lưu ý: sau tham số '-net' là địa chỉ mạng con của mạng bên kia, và sau tham số 'gw' là địa chỉ của card mạng của máy bên kia mà được nối với mạng con đó. Trong phần này chúng tôi sẽ chỉ trình bày kỹ về các tuỳ chọn với file cấu hình options.

2.1 Các tuỳ chọn đặc biệt

Tất cả các tham số cấu hình được xử lý bởi chương trình daemon 'transcryptd'. Các tham số có thể được lấy từ một trong các vị trí và theo thứ tự ưu tiên như sau:

- File tuỳ chọn ngầm định '/etc/transcrypt/options'
- Từ một file tuỳ chọn được chỉ ra bởi tham số '-o file' trên dòng lệnh.
- Các tuỳ chọn đơn được đưa vào từ dòng lệnh.

Điều này có nghĩa là các tham số từ dòng lệnh được dùng thay cho các tham số từ các file, và các tham số từ một file tuỳ chọn được chỉ ra sẽ được dùng thay cho các tham số từ file tùy chọn ngầm định.

Các tuỳ chọn trên có kiểu là một trong các kiểu sau: boolean, integer, string, địa chỉ IP, địa chỉ IP với số cổng. Các tuỳ chọn ngầm định là false và việc chỉ ra chúng nghĩa là đặt chúng thành true. Địa chỉ IP được chỉ ra dưới dạng dấu chấm hoặc tên miền mà có thể phân giải được bằng hàm 'gethostbyname'. Các địa chỉ UDP hoặc TCP được đưa ra theo dạng 'ip:port', mà port là một số hoặc tên phân giải được bằng 'getservbyname'.

Cú pháp của các tuỳ chọn đặc biệt này là 'name=value' trên dòng lệnh, và 'name value' (mỗi tuỳ chọn trên một dòng, không liền nhau, không có ký tự điều khiển, dấu nháy..) trong file tùy chọn.

Vì lý do bảo mật, các file tuỳ chọn phải được đưa ra theo đường dẫn dẫn tuyệt đối, và chỉ được sở hữu bởi người dùng root, bất kỳ nhóm hoặc người dùng khác không có quyền viết, thậm trí là đọc file này (bởi các khoá bí mật đều được ghi trong file này).

2.2 Danh sách tất cả các tham số

(Req=Yêu cầu tham số - Required parameter)

Tên	Kiểu	Req	
device	String	Không	Tên thiết bị TRANSCRYPT. Nếu không đưa ra, hệ thống sẽ lấy một thiết bị còn trống.

debug	Bool		Không chạy ở tiến trình nền, dùng thiết bị lỗi chuẩn stderr thay cho syslog. (độc lập với tuỳ chọn debug của driver nhân)
ipaddr	IP	Có	Địa chỉ IP của card mạng nối với mạng con cần bảo vệ của máy hiện tại.
ptpaddr	IP	Có	Địa chỉ IP của thiết bị của card mạng nối với mạng con cần bảo vệ của máy bên kia.
cttl	UDP	Không	Giá trị truyền thông TTL. Nếu không xác định hoặc bằng 0, chương trình sẽ sử dụng TTL của gói payload. Giá trị ngầm định được thiết lập là 64.
me	UDP	Không	Địa chỉ truyền thông UDP của máy hiện tại, tiếp theo là số cổng được sử dụng để truyền thông được viết sau dấu ':'. Nếu hoặc IP hoặc cổng không được chỉ ra, hệ thống sẽ chọn một địa chỉ bất kỳ.
peer	UDP	Có	Địa chỉ truyền thông UDP và số cổng của phía bên kia. Viết cách nhau dấu ':'.
key	String	Có	Khoá liên kết. Vì lý do bảo mật, khoá liên kết phải được đặt trong file tuỳ chọn. Khoá là 512 bits và được viết dưới dạng hệ cơ số 16.
nokey	Bool		Không mã hoá gì cả, chỉ bao bọc gói tin trong giao thức UDP. Với tùy chọn này, ta không cần đến tùy chọn 'key' (không cần thiết).
tockxc	Int	Không	Thời gian chờ để trao đổi khoá (tính theo giây). Ngầm định là 10.
tokey	Int	Không	Thời gian sống của khoá động. Ngầm định là 600 (10 phút).
ipup	String	Không	Kịch bản để chạy thay cho file /etc/transcrypt/ip-up.
ipdown	String	Không	Kịch bản để chạy thay cho file /etc/transcrypt/ip-down.
arg	String	Không	Tham số để cung cấp cho 'ip-up' và 'ip-down'.
maxerr	Int	Không	Số lỗi lớn nhất cho phép trước khi daemon transcryptd thoát.
tokxts	Int	Không	Thời gian chờ trao đổi nhãn thời gian. Ngầm định là 0 (không sử dụng nhãn thời gian). Có thể thiết lập giá trị cho tuỳ chọn này là 30 để bảo vệ chống lại kiểu tấn công bằng việc truyền lại trao đổi khoá. Tuy nhiên với tuỳ chọn này, đồng hồ của cả 2 đầu phải

			được đồng bộ.
toping	Int	Không	Thời gian chờ để ping. Nếu không nhận được tín hiệu trả lời trong thời gian ping, nó coi đó là một lỗi.
ifconfig	Bool		Yêu cầu lệnh ngoài 'ifconfig' để cấu hình giao diện.

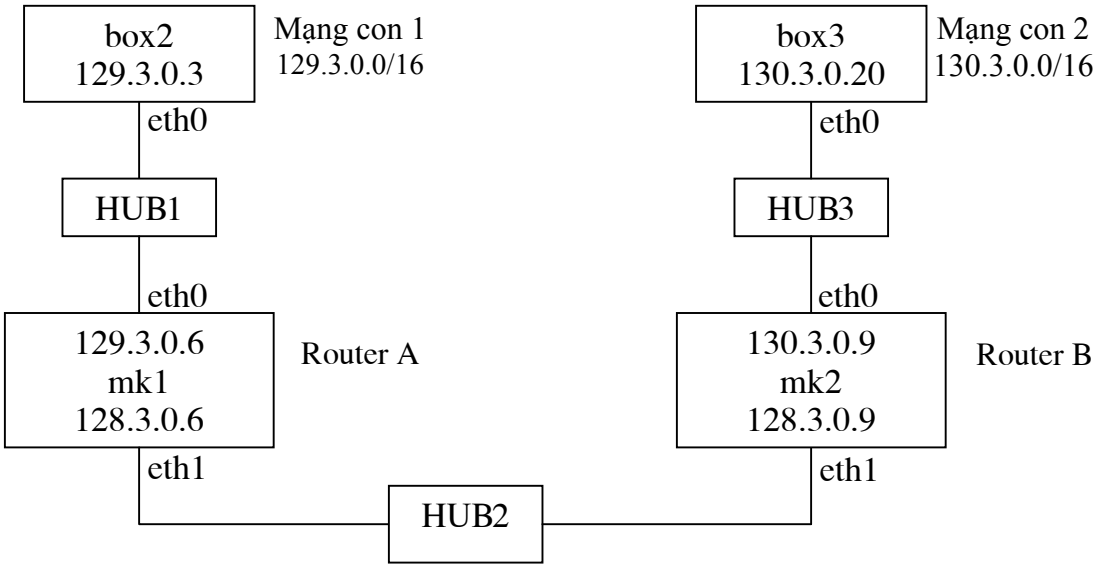
2.3 Lỗi điều khiển trong 'transcryptd'

Khi 'transcrypd' ở máy từ xa bị đóng lại hoặc không khả dụng nữa, 'transcrypd' của máy cục bộ sẽ nhận được lỗi "connection refused". Điều này đã được chứng minh trong thực tế ở một số tình huống, vì vậy có chương trình được thêm vào một tuỳ chọn mở rộng khác: 'maxerr' quyết định 'transcryptd' sẽ bỏ qua bao nhiêu lỗi trước khi nó thoát. Bộ đếm này sẽ được đặt lại mỗi khi kết nối chạy ổn định (chính xác hơn: khi nhận được một gói trao đổi khoá). Giá trị ngầm định cho 'maxerr' là 8. Một giá trị đặc biệt -1 báo cho 'transcrypd' bỏ qua bất kỳ một lỗi về mạng nào. Trong trường hợp này, nó chỉ có thể được đóng lại bằng tay.

3. Cài đặt TRANSCRYPT trên mô hình thử nghiệm

3.1 Cấu hình mạng

Phần mềm TRANSCRYPT đã được biên dịch và thực hiện thử nghiệm trên hệ thống mạng của chúng tôi. Mô hình thử nghiệm được mô tả như hình vẽ ở phần dưới. ở đây chúng tôi lấy mô hình mạng tại phòng thí nghiệm an toàn mạng máy tính của PVKH MM làm ví dụ. Tuỳ thuộc vào mô hình mạng của bạn mà khi cấu hình sẽ có những thay đổi phù hợp.



Theo mô hình trên, hai máy mk1 và mk2 là hai router được nối với nhau qua HUB 2, box2 và box3 lần lượt là 1 máy thuộc mạng con của mk1 và mk2. Chúng ta thiết lập TRANSCRYPT trên hai Router mk1, mk2 để bảo vệ thông tin trao đổi giữa 2

mạng con này. Sau khi thiết lập cấu hình mạng như hình vẽ mô tả ở trên, chúng ta thực hiện các bước sau:

- Sử dụng netconf hoặc chương trình route để đặt địa chỉ gateway ngầm định (default gateway) cho 2 Router mk1 và mk2. Địa chỉ gateway ngầm định của mk1 là 128.3.0.9 (eth1 của mk2) và ngược lại địa chỉ gateway ngầm định của mk2 là 128.3.0.6 (eth1 của mk1).
- Đặt tuỳ chọn net.ipv4.ip_forward = 1 trong file /etc/sysctl.conf hoặc sử dụng lệnh echo 1 > /proc/sys/net/ipv4/ip_forward.
- Khởi động lại dịch vụ mạng bằng lệnh: /etc/rc.d/init.d/network restart.
- Thử 'ping' từ máy box 2 (của mạng con 1) sang máy box 3 (mạng con 2). Nếu thông tức quá trình cấu hình mạng đã thông.
- Sau khi đã đảm bảo rằng cấu hình mạng của chúng đã thông (khi chưa thiết lập Mật mã), chúng ta sẽ bỏ cấu hình gateway ngầm định từ chương trình netconf (hoặc route). Bởi chương trình TRANSCRYPT không sử dụng các gateway ngầm định, nó sẽ tự thêm nó sẽ thêm định tuyến phù hợp (được cấu hình bằng lệnh route trong file /etc/transcrypt/ip-up).

Để thực hiện thử nghiệm với mô hình bảo mật dữ liệu bằng TRANSCRYPT, trên 2 máy box2 và box3 chúng tôi lần lượt cài đặt Camera Tenius (Video CAM III) ver 1.0 và Camera Tenius (Video CAM) ver 1.12 (Bản quyền của hãng CyberLink). Dữ liệu camera được truyền thông qua chương trình Netmeeting của Microsoft Windows trên bản Windows 98. Đặc biệt, chương trình này còn cho phép ta truyền file, chia sẻ màn hình, chat ... giúp ích ta trong quá trình kiểm tra (mang tính trực quan). Dữ liệu cần bảo vệ là dòng dữ liệu truyền thông được truyền qua 2 router là mk1 và mk2.

3.2 Thiết lập các file cấu hình để thực hiện kết nối TRANSCRYPT

Sau khi biên dịch thành công TRANSCRYPT, ta được 1 file kernel module là transcryptcb.o, 1 file daemon 'transcryptd-cb' trong thư mục biên dịch đích và file chương trình 'kex' trong thư mục kex/. Thực hiện việc cài đặt và thiết lập cấu hình như đã trình bày trong mục 1 và 2 ở trên. Dưới đây là các file cấu hình cụ thể đối với cấu hình mạng của chúng tôi.

a. Thiết lập kết nối bằng khoá bí mật

File options của máy mk1 như sau:

```
# Without a "device" line, the device is picked dynamically
device          crypt0
# the peer's IP address
ptpaddr                 130.3.0.9
# our CIPE device's IP address
ipaddr          129.3.0.6
# my UDP address. Note: if you set port 0 here, the system will pick
# one and tell it to you via the ip-up script. Same holds for IP 0.0.0.0.
me              128.3.0.6:6789
```

```
# ...and the UDP address we connect to. Of course no wildcards here.
peer            128.3.0.9:6543
# The static key. Keep this file secret!
# The key is 128 bits in hexadecimal notation.
key             3248fd20adf9c00ccf9ecc2393bbb3e4
```

Và file options trên máy router 2 là:

```
# Without a "device" line, the device is picked dynamically
device          crypt0
# the peer's IP address
ptpaddr                 129.3.0.6
# our CIPE device's IP address
ipaddr          130.3.0.9
# my UDP address. Note: if you set port 0 here, the system will pick
# one and tell it to you via the ip-up script. Same holds for IP 0.0.0.0.
me              128.3.0.9:6789
# ...and the UDP address we connect to. Of course no wildcards here.
peer            128.3.0.6:6543
# The static key. Keep this file secret!
# The key is 128 bits in hexadecimal notation.
key             3248fd20adf9c00ccf9ecc2393bbb3e4
```

File ip-up trên máy mk1 như sau:

```
umask 022
PATH=/sbin:/bin:/usr/sbin:/usr/bin

case `uname -r` in
2.0*)
  # 2.1 and later sets this one by itself.
  #route add -host $5 dev $1
  ;;
esac

# If this becomes our default route...
route add -net 129.3.0.9 dev cryptcb0 gw 129.3.0.9

# just a logging example
now=`date "+%b %d %T"`
echo "$now UP   $*" >> /var/log/transcrypt.log

# Tạo/cập nhật file PID.
echo "$3 $1" >/var/run/transcrypt/${6:-$1}.pid

# Interconnect two 10. subnets through the Internet!
# Assuming $4 is in 10.1 and $5 in 10.2
route add -net 130.3.0.0 netmask 255.255.0.0 gw 128.3.0.9
# Proxy-ARP the peer's address on eth0
```

```
#arp -i eth0 -Ds $5 eth0 pub
exit 0
```

File ip-up trên máy mk2 cũng tương tự như vậy chỉ khác phần địa chỉ route, như sau:
route add -net 130.3.0.9 dev cryptcb0 gw 130.3.0.9

Nạp module transcrypt-cb.o bằng lệnh:
insmod transcrypt-cb.o

Chạy chương trình TRANSCRYPT:
transcryptcb -o /etc/transcrypt/options

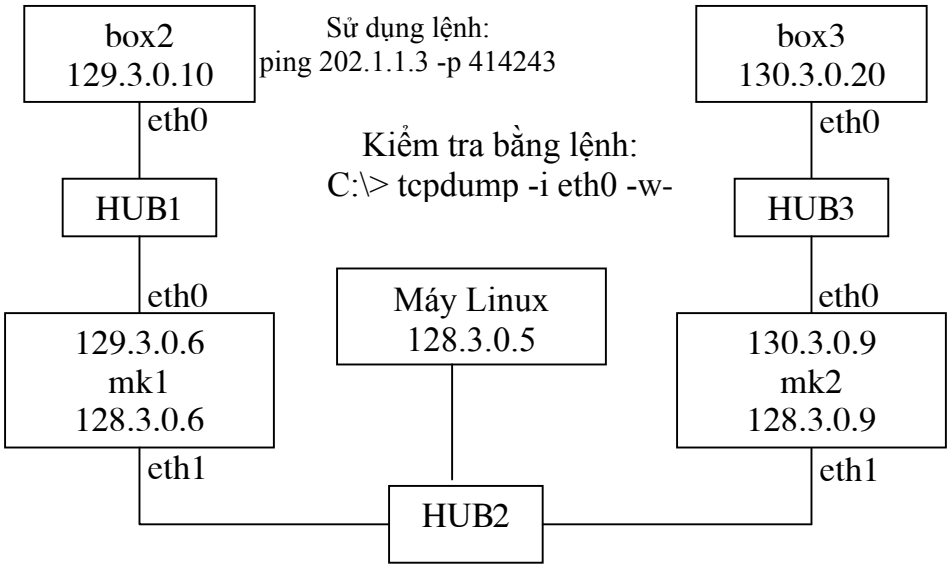
Đến đây, chúng ta có thể chuyển sang bước "Kiểm tra quá trình cài đặt và cấu hình".

b. Transcrypt với trao đổi khoá động KEX

Để thực hiện được quá trình trao đổi khoá tự động thì chúng ta cần phải cấu hình các bên tham gia kết nối như được mô tả trong chương 5. Trình thực hiện việc trao đổi khoá là kex. Sau khi thực hiện trao đổi khoá thành công, chương trình kex thực hiện gọi chương trình transcryptd-cb với một file tuỳ chọn mới chứa khoá mới được thoả thuận. Đến đây, ta có thể chuyển sang bước "kiểm tra quá trình cài đặt và cấu hình".

c. Kiểm tra quá trình cài đặt và cấu hình

- Ta sử dụng lệnh 'ifconfig' để xem đã có thêm một thiết bị mạng ảo 'cryptcb0' chưa. Nếu chưa có, tức chương trình 'transcryptd-cb' chưa thể chạy được. Bạn có thể xem các thông điệp nhật ký hệ thống (/var/log/message và /var/log/transcrypt.log) để tìm hiểu về nguyên nhân gây là lỗi.
- Trên mỗi máy thuộc mạng con bên này, ta 'ping' sang 1 máy của mạng bên kia để kiểm tra xem mạng đã thông chưa. Nếu mạng chưa thông, hãy kiểm tra giá trị ip_forward trong file /proc/sys/net/ipv4 có là 1 không, sau đó kiểm tra lại quá trình cấu hình TRANSCRYPT.
- Dùng chương trình tcpdump để kiểm tra thông tin được trao đổi giữa mk1 và mk2 xem đã được mã hoá chưa. Tại box2 dùng lệnh 'ping 130.3.0.20 -p 414243, lệnh ping với tham số -p 414243 sẽ gửi dữ liệu là 3 ký A (41), B(42) và C(43) sang máy 130.3.0.20. Tại máy Linux (128.3.0.5), ta kiểm tra bằng lệnh tcpdump -i eth0 -w-, thông tin hiện lên màn hình cho biết đã mã hay không.



Nếu các bước kiểm tra trên đều tốt đẹp, thì hệ thống bảo mật của ta đã hoạt động. Khi này dữ liệu giao tiếp trên 2 mạng con đã được bảo vệ trên kênh truyền. Ta có thể dùng chương trình Netmeeting để truyền dữ liệu (Camera, file, share, chat, ...) qua 2 mạng con đã được bảo vệ này để kiểm tra hệ thống.

B. PHẦN MỀM IP-CRYPTO

Chương 1
Giải pháp bảo mật của IP-CRYPTO

1. Quản lý các gói tin mạng trong nhân Linux

Linux quản lý các gói tin mạng được gửi đi và nhận về theo một phương thức thống nhất. Đó chính là việc cung cấp vùng hàng đợi bộ nhớ để quản lý gói tin mạng. Độ dài của vùng bộ nhớ đệm (buffer) được quản lý một cách khá mềm dẻo, điều đó tạo điều kiện cho chúng ta có thể dễ dàng thay đổi độ dài vùng buffer theo chủ ý. Một gói tin mạng được nhân Linux quản lý bằng một cấu trúc sk_buff. Cấu trúc này được khai báo trong file skbuff.h như sau:

```
struct sk_buff {
      struct sk_buff    * next;      /* Next buffer in list */
      struct sk_buff    * prev;      /* Previous buffer in list */
      struct sk_buff_head * list;    /* List we are on      */
      struct sock *sk;               /* Socket we are owned by */
      struct timeval    stamp;       /* Time we arrived */
      struct device     *dev; /* Device we arrived on/are leaving by */

      /* Transport layer header */
      union
      {
            struct tcphdr     *th;
            struct udphdr     *uh;
            struct icmphdr    *icmph;
            struct igmphdr    *igmph;
            struct iphdr      *ipiph;
            struct spxhdr     *spxh;
            unsigned char     *raw;
      } h;

      /* Network layer header */
      union
      {
            struct iphdr      *iph;
            struct ipv6hdr    *ipv6h;
            struct arphdr     *arph;
            struct ipxhdr     *ipxh;
            unsigned char     *raw;
      } nh;

      /* Link layer header */
      union
      {
            struct ethhdr     *ethernet;
            unsigned char     *raw;
      } mac;

      struct  dst_entry *dst;
```

```
      char        cb[48];

      unsigned int      len;        /* Length of actual dat */
      unsigned int      csum;       /* Checksum */
      volatile char     used;
      unsigned char     is_clone,   /* We are a clone */
      cloned,
      pkt_type,                     /* Packet class   */
      ip_summed;                    /* Driver fed us an IP checksum*/
      __u32 priority;               /* Packet queueing priority */
      atomic_t    users;          /* User count-see datagram.c,tcp.c */
      unsigned short    protocol;   /* Packet protocol from driver. */
      unsigned short    security;   /* Security level of packet */
      unsigned int      truesize;   /* Buffer size */

      unsigned char     *head;      /* Head of buffer */
      unsigned char     *data;      /* Data head pointer */
      unsigned char     *tail;      /* Tail pointer   */
      unsigned char     *end;       /* End pointer    */
      void        (*destructor)(struct sk_buff *);
#ifdef CONFIG_IP_FIREWALL
        __u32           fwmark;                 /* Label made by
fwchains, used by pktsched    */
#endif
#if defined(CONFIG_SHAPER) || defined(CONFIG_SHAPER_MODULE)
      __u32       shapelatency;     /* Latency on frame */
      __u32       shapeclock;       /* Time it should go out */
      __u32       shapelen;         /* Frame length in clocks */
      __u32       shapestamp;       /* Stamp for shaper    */
      __u16       shapepend;        /* Pending */
#endif

#if defined(CONFIG_HIPPI)
      union{
            __u32 ifield;
      } private;
#endif
};
```

Ta nhận thấy, trong cấu trúc này có con trỏ chỉ tới đầu frame (khung tin), và cuối frame. Ở mỗi tầng tương ứng của mô hình mạng sẽ có con trỏ chỉ tới phần header của tầng đó. Ví dụ, ở tầng datalink thì có cấu trúc struct ethhdr *ethernet, ở tầng mạng có cấu trúc struct iphhdr *iph . . . Mỗi gói tin đến và đi đều gắn liền với một giao diện mạng tương ứng, giao diện này được chứa trong trường cấu trúc struct device *dev. Ngoài ra, Linux còn có hỗ trợ các hàm lập trình kernel API dùng để cung cấp các phương thức dịch chuyển con trỏ trên struct sk_buff, và các phương thức cung cấp thêm bộ nhớ cho phần đầu, phần cuối. . .

2. Kỹ thuật tạo card mạng ảo và cách gửi gói tin qua card mạng ảo

Khi chúng ta cài đặt các giao diện mạng, các giao diện này được đăng ký vào nhân thường thì các giao diện mạng được nạp vào nhân dưới dạng một module khả nạp.

Ví dụ, giao diện mạng ethernet 3c509, dmfe . . . tương ứng với nó là các module 3c509.o, dmfe.o . . . Khi các module này được nạp tương ứng với nó là các giao diện đại diện trong nhân là eth0, eth1 . . . Ngoài việc quản lý các giao diện vật lý, nhân Linux còn hỗ trợ kiểu giao diện mạng ảo. Để có giao diện mạng ảo, chúng ta cần phải viết một module để giả lập cạc mạng vật lý thật. Sở dĩ chúng ta làm được như vậy là do nhân Linux quản lý giao diện mạng bằng một cấu trúc struct device chung. Cấu trúc này được khai báo trong file include/linux/netdevice.h:

```
struct device
{
      /*
      * This is the first field of the "visible" part of this structure
      * (i.e. as seen by users in the "Space.c" file).  It is the name
      * the interface.
      */
      char              *name;

      /*
       *    I/O specific fields
       *    FIXME: Merge these and struct ifmap into one
       */
      unsigned long           rmem_end;   /* shmem "recv" end     */
      unsigned long           rmem_start; /* shmem "recv" start   */
      unsigned long           mem_end;    /* shared mem end */
      unsigned long           mem_start;  /* shared mem start     */
      unsigned long           base_addr;  /* device I/O address   */
      unsigned int            irq;        /* device IRQ number    */

      /* Low-level status flags. */
      volatile unsigned char  start;            /* start an operation
      */
      /*
       * These two are just single-bit flags, but due to atomicity
       * reasons they have to be inside a "unsigned long". However,
       * they should be inside the SAME unsigned long instead of
       * this wasteful use of memory..
       */
      unsigned long           interrupt;  /* bitops.. */
      unsigned long           tbusy;            /* transmitter busy */

      struct device           *next;

      /* The device initialization function. Called only once. */
      int               (*init)(struct device *dev);
      void              (*destructor)(struct device *dev);

      /* Interface index. Unique device identifier    */
      int               ifindex;
      int               iflink;

      /*
       *    Some hardware also needs these fields, but they are not
       *    part of the usual set specified in Space.c.
       */
```

```
      unsigned char           if_port;    /* Selectable AUI, TP,..*/
      unsigned char           dma;        /* DMA channel          */

      struct net_device_stats* (*get_stats)(struct device *dev);
      struct iw_statistics* (*get_wireless_stats)(struct device *dev);

      /*
       * This marks the end of the "visible" part of the structure. All
       * fields hereafter are internal to the system, and may change at
       * will (read: may be cleaned up at will).
       */

      /* These may be needed for future network-power-down code. */
      unsigned long     trans_start; /* Time (in jiffies) of last Tx*/
      unsigned long           last_rx;    /* Time of last Rx      */

      unsigned short          flags;      /* interface flags (a la BSD)
      */
      unsigned short          gflags;
      unsigned          mtu;  /* interface MTU value        */
      unsigned short          type; /* interface hardware type    */
      unsigned short    hard_header_len;  /* hardware hdr length  */
      void              *priv;      /* pointer to private data    */

      /* Interface address info. */
      unsigned char           broadcast[MAX_ADDR_LEN]; /*hw bcast add*/
      unsigned char           pad;   /* make dev_addr aligned to 8 bytes
*/
      unsigned char           dev_addr[MAX_ADDR_LEN]; /* hw address */
      unsigned char           addr_len;   /* hardware address length */

      struct dev_mc_list      *mc_list;   /* Multicast mac addresses */
      int               mc_count;   /* Number of installed mcasts */
      int               promiscuity;
      int               allmulti;

      /* For load balancing driver pair support */

      unsigned long           pkt_queue;  /* Packets queued */
      struct device           *slave;           /* Slave device
      */

      /* Protocol specific pointers */

      void              *atalk_ptr; /* AppleTalk link       */
      void              *ip_ptr;    /* IPv4 specific data   */
      void              *dn_ptr;        /* DECnet specific data */

      struct Qdisc            *qdisc;
      struct Qdisc            *qdisc_sleeping;
      struct Qdisc            *qdisc_list;
      unsigned long           tx_queue_len; /* Max frames per queue
allowed */

      /* Bridge stuff */
      int               bridge_port_id;
```

```
      /* Pointers to interface service routines.      */
      int               (*open)(struct device *dev);
      int               (*stop)(struct device *dev);
      int               (*hard_start_xmit) (struct sk_buff *skb,
                                            struct device *dev);
      int               (*hard_header) (struct sk_buff *skb,
                                        struct device *dev,
                                        unsigned short type,
                                        void *daddr,
                                        void *saddr,
                                        unsigned len);
      int               (*rebuild_header)(struct sk_buff *skb);
#define HAVE_MULTICAST
      void              (*set_multicast_list)(struct device *dev);
#define HAVE_SET_MAC_ADDR
      int               (*set_mac_address)(struct device *dev,
                                           void *addr);
#define HAVE_PRIVATE_IOCTL
      int               (*do_ioctl)(struct device *dev,
                                    struct ifreq *ifr, int cmd);
#define HAVE_SET_CONFIG
      int               (*set_config)(struct device *dev,
                                        struct ifmap *map);
#define HAVE_HEADER_CACHE
      int               (*hard_header_cache)(struct neighbour *neigh,
                                             struct hh_cache *hh);
      void              (*header_cache_update)(struct hh_cache *hh,
                                               struct device *dev,
                                               unsigned char *  haddr);
#define HAVE_CHANGE_MTU
      int               (*change_mtu)(struct device *dev, int new_mtu);

      int               (*hard_header_parse)(struct sk_buff *skb,
                                             unsigned char *haddr);
      int               (*neigh_setup)(struct device *dev, struct
neigh_parms *);
      int               (*accept_fastpath)(struct device *, struct
dst_entry*);

#ifdef CONFIG_NET_FASTROUTE
      /* Really, this semaphore may be necessary and for not fastroute
code;
       */
      int               tx_semaphore;
#define NETDEV_FASTROUTE_HMASK 0xF
      /* Semi-private data. Keep it at the end of device struct. */
      struct dst_entry  *fastpath[NETDEV_FASTROUTE_HMASK+1];
#endif
};
```

Trong cấu trúc device trường char *name chứa tên của giao diện (ví dụ, eth0, eth1 ...), và chứa một số hàm được thực thi trên giao diện đó bao gồm như hàm thực hiện kết gán địa chỉ vật lý int *set_mac_address(struct device *dev, void *addr), hàm mở thiết bị int *open(struct device *dev), hàm đóng thiết bị int *close(struct

device *dev), hàm cung cấp giao diện giữa người dùng và nhân int *do_ioctl(struct device *dev , struct ifreq *ifr, int cmd). . . Và chúng ta đặc biệt quan tâm tới hàm gửi gói tin int *hard_start_xmit(struct sk_buff *skb, struct device *dev).

Hàm int *hard_start_xmit(struct sk_buff *skb, struct device *dev) sẽ thực hiện gửi gói tin được chứa trong buffer skb qua giao diện mạng dev. Đây là hàm gửi gói tin ra giao diện vật lý thực sự. Chúng ta có thể thực hiện các công việc cần thiết ở trong hàm này như bọc thêm địa chỉ IP, thêm bớt dữ liệu trong gói tin mạng (Gói tin bao gồm phần dữ liệu và phần header). . . Chúng ta sẽ thực sự khảo sát kỹ hàm này trong phần gửi gói tin đi của phần mềm IP-Cryplink-3.0.

Sau khi đã kết gán các tham số cho cấu trúc giao diện mạng, chúng ta dùng hàm kernel API register_netdev(struct device *dev) để thực hiện đăng ký giao diện mạng trong nhân.

Chúng ta dùng hàm unregister_netdev(struct device *dev) để loại bỏ giao diện mạng khi không sử dụng đến nó nữa.

3. Nhận gói tin mạng trong nhân Linux

Ở phần trên chúng ta đã khảo sát cách tạo ra giao diện mạng ảo để gửi gói tin đi trong nhân Linux. Ở phần này chúng ta sẽ đề cập tới vấn đề nhận gói tin gửi đến trong nhân Linux như thế nào. Như chúng ta đã biết, quá trình nhận gói tin sẽ được thực hiện từ dưới lên trên theo mô hình tầng mạng. Đầu tiên giao diện vật lý sẽ thực hiện nhận gói tin. Sau đó gói tin được chuyển lên cho tầng liên kết dữ liệu (data link) và tiếp đến là tầng IP (nếu họ giao thức chúng ta dùng là TCP/IP) ... Ở mỗi tầng thì nhân Linux cung cấp các hàm xử lý gói tin khác nhau. Khi lên ở tầng cao hơn thì phần thông tin header của tầng mạng dưới bị loại bỏ.

Một điều đặc biệt thú vị là khi nhận gói tin ở giao diện vật lý, Linux cung cấp giải pháp xử lý gói tin tuỳ thuộc vào chỉ số giao thức gói tin được gửi. Cấu trúc struct inet_protocol trong file include/net/protocol.h chứa chỉ số giao thức của gói tin và các hàm xử lý đối với gói tin có giao thức này:

```
/* This is used to register protocols. */
struct inet_protocol
{
      int         (*handler)(struct sk_buff *skb, unsigned short len);
      void        (*err_handler)(struct sk_buff *skb, unsigned char *dp,
int len);
      struct inet_protocol    *next;
      unsigned char           protocol;
      unsigned char           copy:1;
      void              *data;
```

```
        const char        *name;
};
```

Trong cấu trúc trên thì trường unsigned char protocol chính là trường chứa giao thức của gói tin cần xử lý, còn con trỏ hàm int (*handler)(struct sk_buff *skb, unsignecd short len) là hàm xử lý gói tin có chỉ số giao thức là protocol.

Nhân Linux hỗ trợ hàm void inet_add_protocol(struct inet_protocol *proto) để đăng ký một cấu trúc inet_protocol đã được khởi tạo vào nhân, và việc loại bỏ cấu trúc inet_protocol đã có sẵn trong nhân được thực hiện bởi hàm inet_del_protocol(struct inet_protocol *proto).

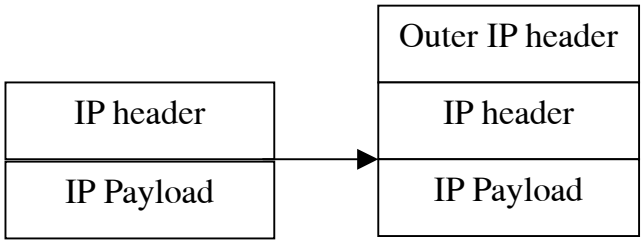
Như vậy, trong chương này chúng ta đã khảo sát xong các kỹ thuật gửi và nhận gói tin được thực hiện trong nhân Linux. Trong chương II chúng ta sẽ xem xét giải pháp can thiệp cụ thể trong việc gửi và nhận gói tin của phần mềm IP-Crypto.

4. Các mô hình bảo mật gói tin ở tầng IP trong IP-Crypto

4.1 Mô hình hoạt động với sự tạo lập đường hầm trong IP-Crypto (tunnel mode)

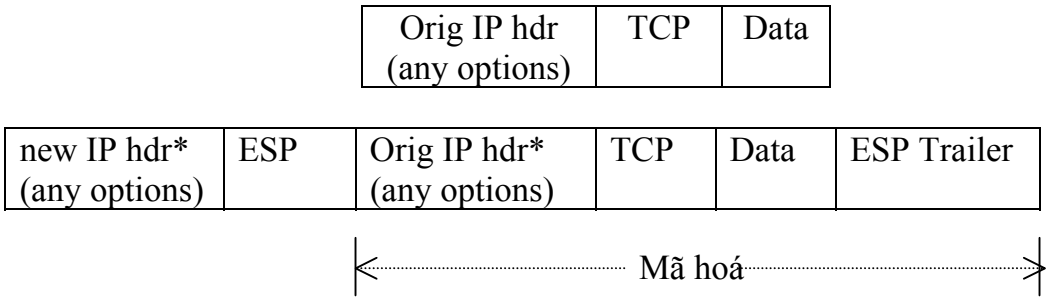
Khi một gói tin chuyển đi từ SUBNET1 sang SUBNET2 sẽ được máy GATEWAY1 mã hoá và bọc gói tin, sang bên máy GATEWAY2 sẽ thực hiện việc giải mã, loại bỏ phần IP thêm vào, để tạo ra gói tin nguyên bản ban đầu.

Mô hình tạo lập tunnel (đường hầm) giữa hai gateway bảo mật: Thực chất của việc tạo lập tunnel đó chính là chúng ta bọc thêm phần IP header mới cho gói tin đi giữa hai máy gateway:



Gói tin đi từ Máy1 sang Máy2 sẽ được Encapsulator thêm vào gói tin một IP header mới và gửi đi sang cho Decapsulator. Khi Decapsulator nhận được gói tin đã được bọc thêm phần IP header mới thì nó sẽ thực hiện việc loại bỏ phần IP header mới đi và trả lại dạng gói tin nguyên bản ban đầu để gửi sang Máy2.

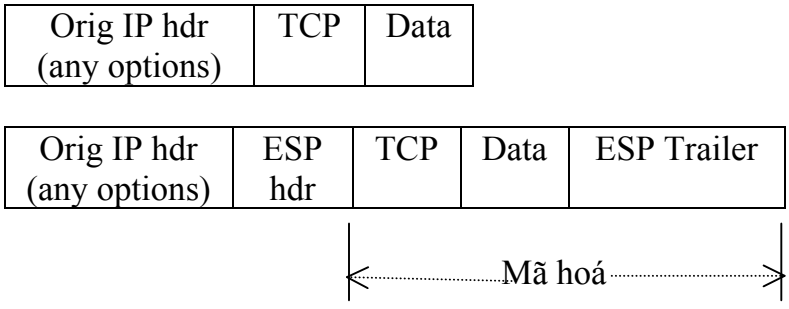
Dựa vào kỹ thuật lập đường hầm có trong Linux, chúng ta có thể xây dựng chương trình bảo mật gói IP mà toàn bộ gói tin được truyền đi giữa hai máy gateway sẽ được bảo mật kể cả phần IP header ban đầu của gói tin. Mô hình của gói tin trước và sau khi được bảo mật như sau:



4.2 Bảo mật ở tầng IP cho phiên truyền thông giữa hai máy (Transport mode)

Ở mô hình bảo mật gói IP có thiết lập đường hầm dùng để bảo mật giữa hai mạng riêng ở hai đầu máy gateway (Mô hình mạng riêng ảo VPN – Virtual Private Network). Nhưng chúng ta cũng có thể bảo mật phiên truyền thông giữa hai máy mà không cần việc thiết lập đường hầm (tunnel). Ở trường hợp này định dạng IP header của gói tin được gửi đi không bị thay đổi (Không thực hiện mã hoá phần IP header), mà chỉ có nội dung gói tin được mã hoá. Sang bên máy nhận sẽ thực hiện việc giải mã nội dung gói tin để lấy lại nguyên dạng gói tin ban đầu chưa mã hóa.

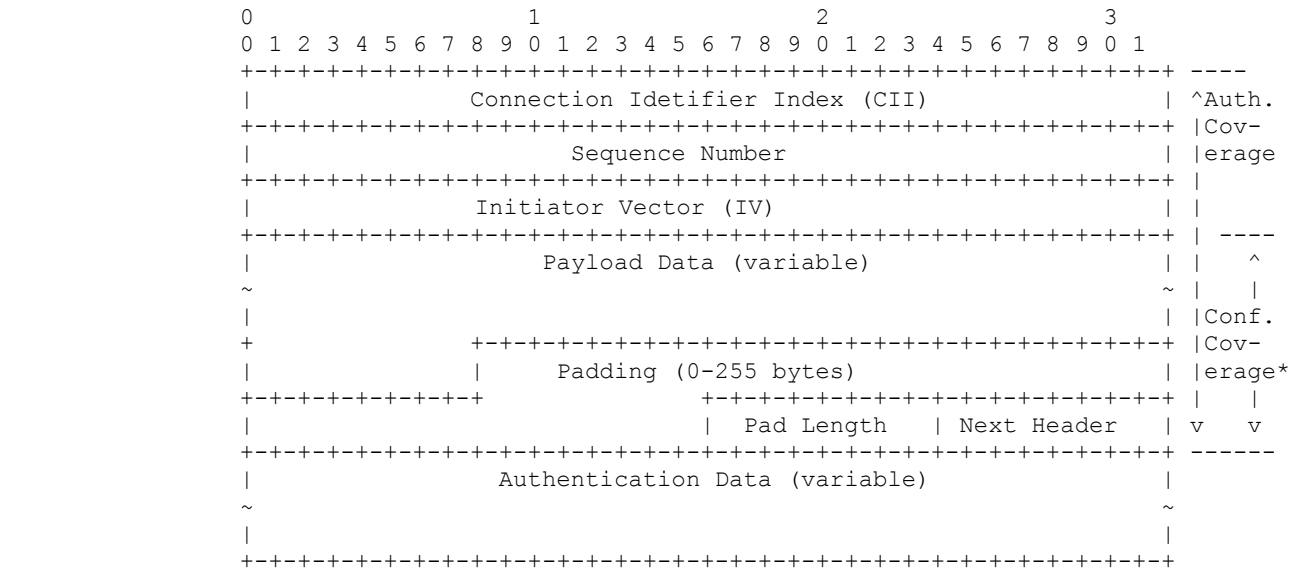
Mô hình được thể hiện như sau:



4.3 Định dạng gói tin đóng viên ESP (Encapsulating Security Payload Packet Format)

Ở cả hai mô hình bảo mật trên thì sự khác nhau duy nhất đó là việc thiết lập đường hầm khi bảo mật. Một định dạng gói tin được sử dụng cho cả hai trường hợp đó chính là gói tin đóng viên ESP (Encapsulating Security Payload). Gói tin đóng viên ngoài phần dữ liệu được bảo mật (được mã hoá) còn bao gồm các trường thông tin khác dùng để bảo đảm sự đồng bộ thông suốt của quá trình mã hoá và giải mã.

Gói tin sau khi được mã hoá gói là gói tin ESP (Encapsulation Security Payload) sẽ có định dạng như sau:



- ✓ Connection Identifier Index

  CII là giá trị 32-bít, trong sự kết hợp với địa chỉ IP đích và giao thức an toàn (ESP), định danh duy nhất cho connection (cặp kết nối) sử dụng cho gói tin này. Trên một máy gateway có thể có nhiều connection cùng tồn tại, một connection được sử dụng để bảo mật cho một cặp mạng riêng hay cho giao tiếp giữa hai máy cụ thể nào đó. Vì thế, chúng ta sử dụng CII để chọn connection khi nhận được gói tin ESP. Phần này sẽ được trình bày cụ thể hơn trong phần nhận gói tin gửi đến.

- ✓ Sequence Number (Chuỗi số)

  Là trường số không dấu 32-bit, giá trị của trường này được tăng lên sau mỗi packet được gửi đi. Việc sử dụng trường này nhằm mục đích chống tấn công lặp lại. Với gói tin có trường chuỗi số lặp lại trường chuỗi số của gói tin trước hoặc giá trị của trường này quá bé so với khoảng giá trị trường chuỗi số hiện tại thì gói tin đó không được chấp nhận.

- ✓ Initiator Vector (IV)
  Đây là phần dữ liệu của véc tơ khởi điểm của từng gói tin. Để đảm bảo được việc giải mã gói tin mà không bị ảnh hưởng bởi việc lặp gói tin, mất gói tin thì trên mỗi gói tin bảo mật phải chứa một véc tơ khởi điểm cho chính nó.

- ✓ Payload Data (Dữ liệu gói tin)
  Đây là phần chứa dữ liệu của gói tin mang đi. Trường Payload Data là bắt buộc và nó là số nguyên dương độ dài các byte..

- ✓ Padding (For Encryption) (Thêm dữ liệu cho mã hoá).
  Đây là phần dữ liệu thêm vào cho chẵn khối đầu vào mã hoá. Trường này áp dụng cho hàm mã hoá là mã khối ở mode ECB, hoặc CBC.

- ✓ Pad Length
  Trường Pad Length chỉ ra số byte được thêm vào mà đứng ngay trước nó. Giá trị hợp lệ là từ 0 – 255. Nếu giá trị của trường này là 0 điều đó có nghĩa là không có phần dữ liệu được thêm vào.

- ✓ Next Header
  Next Header là trường 8-bít chỉ ra kiểu dữ liệu chứa trong trường Payload Data. Trường này chỉ ra kiểu gói tin được sử dụng ở đây chính là kiểu gói tin ESP (IPPROTO_ESP).

- ✓ Authentication Data (Dữ liệu xác thực)
  Dữ liệu xác thực là trường chứa ICV (Intergrity Check Value) được tính trên toàn bộ gói tin ESP trừ phần dữ liệu xác thực. Độ dài của trường được xác định bởi hàm xác thực được lựa chọn. Ở đây chúng tôi sử dụng hàm xác thực là hàm băm MD5 với đầu ra được chọn là 96 bít.

Nằm ở đầu trong phần dữ liệu gói tin (sau IP header) là các tham số về số hiệu bộ tham số an toàn áp dụng cho gói tin (CII) là trường *esp*_cii, số hiệu của gói tin *esp*_rpl, và véc tơ khởi điểm esp_iv áp dụng cho gói tin. Cả 3 tham số này được chứa trong cấu trúc struct esp ở trong file ipsec_esp.h như sau:

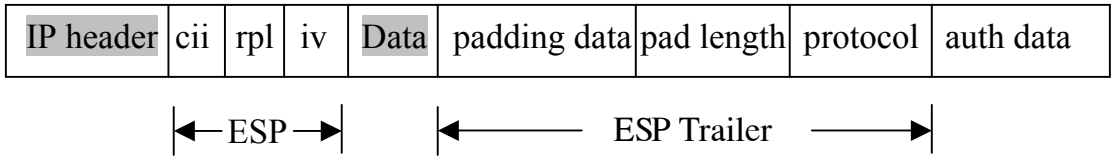
```
struct esp
{
      _ _ u32 esp_cii;  /* Security Parameters Index */
      _ _ u32 esp_rpl;  /* Replay counter */
      _ _ u8 esp_iv[EMT_ESPBLOCKCIPHER_IV_SZ];  /* iv */
}
```

Mô hình gói tin khi thêm các tham số bộ tham số an toàn cho gói tin cii, chỉ số gói tin rpl, và véc tơ khởi điểm là:

IP header	cii	rpl	iv	Data

Phần dữ liệu thêm vào để chẵn khối mã hoá được gán vào theo chuỗi các byte liên tiếp. Sau phần dữ liệu padding là byte báo độ dài dữ liệu thêm vào (trường Pad length) gồm có bao nhiêu byte. Tiếp theo là byte báo giao thức ESP được sử dụng, sau byte này là 12 byte (96 bít) dữ liệu xác thực cho gói tin sinh ra từ hàm MD5. (Tất cả các công việc gắn các phần dữ liệu này được trình bày cụ thể ở phần 1.5 và 1.6). Sở dĩ chúng ta phân biệt được giữa gói tin bảo mật và gói tin không bảo mật là nhờ vào chỉ số giao thức gói tin là ESP hay không phải là ESP.

Mô hình gói tin IP được bảo mật gửi đi trên đường truyền như sau:



5. Quá trình gửi gói tin trong IP-Crypto

Phần mềm IP-Cryplink-3.0 gửi gói tin được bảo mật thông qua cạc mạng ảo được dựng sẵn. Chỉ những gói tin nào được bảo mật mới được cạc mạng ảo này xử lý. Một cạc mạng được đại diện bởi một cấu trúc device. Trong cấu trúc device, có chứa các tham số liên quan đến thiết bị như tên giao diện, địa chỉ vào ra của thiết bị phần cứng, ngắt của phần cứng, và các hàm khi khởi tạo. Với các con trỏ hàm trong cấu trúc device, sẽ trỏ đến những hàm thực hiện các công việc cần thiết khi giao diện mạng được khởi tạo và hoạt động. Trong các con trỏ hàm đó thì hàm thực hiện việc gửi gói tin đi đó chính là con trỏ dev->hard_start_xmit. Như vậy, ta có thể thiết lập một hàm gửi dữ liệu mà ta có thể thực hiện các bước xử lý gói tin trước khi gửi đi. Các công việc khi khởi tạo một giao diện mạng thực hiện ở hàm ipsec_tunnel_init(struct device *dev) trong tệp ipsec_tunnel.c:

```
int
ipsec_tunnel_init(struct device *dev)
{
      int i;

      . . .

      /* Add our tunnel functions to the device */
      dev->open         = ipsec_tunnel_open;
      dev->stop         = ipsec_tunnel_close;
      dev->hard_start_xmit    = ipsec_tunnel_start_xmit;
      dev->get_stats          = ipsec_tunnel_get_stats;
```

```
        . . .

        dev->set_multicast_list = NULL;
        dev->do_ioctl           = ipsec_tunnel_ioctl;
        dev->hard_header        = NULL;
        dev->rebuild_header     = NULL;
        dev->set_mac_address    = NULL;
        dev->header_cache_update= NULL;
        dev->neigh_setup        = ipsec_tunnel_neigh_setup_dev;
        dev->hard_header_len    = 0;
        dev->mtu                = 0;
        dev->addr_len           = 0;
        dev->type               = ARPHRD_TUNNEL;
        dev->tx_queue_len       = 10;       /* Small queue */
        memset(dev->broadcast,0xFF, ETH_ALEN);

        /* New-style flags. */
        dev->flags        = IFF_NOARP /* 0 */ /* Petr Novak */;
        dev_init_buffers(dev);
        return 0;
}
```

Để đăng ký một card mạng trong nhân Linux, chúng ta gọi hàm register_netdev(struct device *):

Trong hàm ipsec_tunnel_init_devices() ở tệp ipsec_tunnel.c, thực hiện việc đăng ký card mạng trong nhân như sau:

```
static struct device dev_ipsec0 = {
        "ipsec0\0 ",
        ...,
        ipsec_tunnel_probe
        };

        if (register_netdev(&dev_ipsec0) != 0) /* Gọi hàm register_netdev
để đăng ký cạc mạng */
                return -EIO;
```

Khi hàm ipsec_tunnel_init được gọi thì trong nhân sẽ có một thiết bị mạng ảo ipsec được đại diện bởi một cấu trúc device. Khi gói tin được gửi qua card mạng ảo ipsec thì hàm dev->hard_start_xmit() sẽ thực hiện các công việc cần thiết để gửi gói tin đi. Theo đó, con trỏ hàm dev->hard_start_xmit() chỉ tới hàm ipsec_tunnel_start_xmit().

Hàm ipsec_tunnel_start_xmit( ) thực hiện các công việc mã hoá gói tin, tính dữ liệu xác thực, bọc thêm địa chỉ IP mới (nếu trong chế độ tunnel) . . . Tuy nhiên, ở gateway bảo mật dùng IPSEC thì không phải mọi gói tin đi qua cạc mạng ảo ipsec đều được bảo mật mà phải những gói tin nào thoả mãn điều kiện địa chỉ IP nguồn và đích của nó nằm trong cặp địa chỉ mạng riêng ảo được diện bảo mật. Nếu không thoả mãn thì hàm ipsec_tunnel_start_xmit( ) sẽ không thực hiện bất cứ thao tác nào đối với những loại gói tin này.

Đoạn trình xác định có bộ tham số an toàn cho gói tin hiện tại hay không:

```
/*
 * First things first -- look us up in the erouting tables.
 */
matcher.sen_len = sizeof (struct sockaddr_encap);
matcher.sen_family = AF_ENCAP;
matcher.sen_type = SENT_IP4;
matcher.sen_ip_src.s_addr = iph->saddr;
matcher.sen_ip_dst.s_addr = iph->daddr;
```

/* Tìm kiếm bộ tham số an toàn cho gói tin */

```
er = ipsec_findroute(&matcher);
if(er) {
      outgoing_cnid = er->er_cnid;
}
```

/* Lấy bộ tham số an toàn tương ứng */

```
spin_lock_irqsave(&tdb_lock, tdb_flags);
tdbp = gettdb(&outgoing_cnid);
```

Một gói tin IP được mã hoá bằng hàm mã khối MK1 trong mode CBC, thì phần đầu của nó cần phải khai báo véc tơ khởi điểm ban đầu. Cùng với nó là các thông tin CII (Connection Information Index) khai báo gói tin thuộc connection nào, chỉ số số thứ tự gói tin gửi đi để chống lại kiểu tấn công lặp lại. Chúng ta cần phải biết độ dài của các trường này để thực hiện việc cung cấp khoảng trống cho buffer skb. Đoạn trình xác định độ dài của các trường:

```
switch(tdbp->tdb_cnid.proto) {
      case IPPROTO_ESP:
            switch(tdbp->tdb_encalg) {
                  case ESP_BLOCKCIPHER:
                        headroom += sizeof(struct esp);
                        break;
                    . . .
                  }
      ...
}
```

Nếu trong trường hợp thiết lập tunnel thì ở phần đầu của cấu trúc buffer skb chúng ta còn phải dành phần trống cho phần IP header mới thêm vào:

```
case IPPROTO_IPIP:
      headroom += sizeof(struct iphdr);
      break;
```

Ở phần cuối của buffer skb, vì độ dài gói tin là bất kỳ, muốn áp dụng được hàm mã khối MK1 ở mode CBC thì chúng ta cần phải thực hiện thêm phần dữ liệu chẵn khối (padding data). Đoạn trình tính phần dữ liệu thêm vào cho chẵn khối là:

```
tailroom += ((BLOCKCIPHER_DATA_SIZE - ((pyldsz + 2 * sizeof(unsigned
char)) % BLOCKCIPHER_DATA_SIZE)) % BLOCKCIPHER_DATA_SIZE) + 2;
```

Mặt khác, vì dữ liệu xác thực gồm 12 byte đặt ở cuối gói tin nên chúng ta cũng cần cung cấp vùng nhớ cho nó:

```
case AH_MD5:
      authlen = AHHMAC_HASHLEN;
      break;
```

Tổng độ dài cần cung cấp ở cuối gói tin bao gồm phần dữ liệu thêm vào chẵn khối và phần dữ liệu xác thực:

```
tailroom += authlen;
```

Để đảm bảo cho mỗi gói tin được mã hoá và giải mã đúng. Tránh trường hợp luồng gói tin đến đích không đúng thứ tự, hoặc có sự mất mát gói tin trên đường truyền sẽ ảnh hưởng đến các gói tin sau đó. Thì ở mỗi gói tin phải chứa véc tơ khởi điểm của gói tin đó. Đoạn trình thực hiện việc lấy và gán véc tơ khởi điểm trong hàm ipsec_tunnel_start_xmit() trong tệp ipsec_tunnel.c:

```
case ESP_BLOCKCIPHER:
      iv[0] = *((__u32*)&(espp->esp_iv)    ) =
              ((__u32*)(tdbp->tdb_iv))[0];
      iv[1] = *((__u32*)&(espp->esp_iv) + 1) =
              ((__u32*)(tdbp->tdb_iv))[1];
      iv[2] = *((__u32*)&(espp->esp_iv) + 2) =
              ((__u32*)(tdbp->tdb_iv))[2];
      iv[3] = *((__u32*)&(espp->esp_iv) + 3) =
              ((__u32*)(tdbp->tdb_iv))[3];
```

Trong chương trình chúng tôi có sử dụng một trường chứa số thứ tự của gói tin. Số thứ tự này đúng chính bằng số gói tin được gửi đi. Việc sử dụng trường số thứ tự này sẽ chống lại được kiểu tấn công lặp lại:

```
espp->esp_rpl = htonl(++(tdbp->tdb_replaywin_lastseq));
```

Bởi vì ở trên hai gateway có thể tồn tại nhiều connection khác nhau, điều đó có nghĩa là có nhiều gói tin được bảo mật giữa các cặp mạng ảo khác nhau. Ở bên nhận, để phân biệt được connection nào được sử dụng cho gói tin đến thì bên nhận phải dựa vào chỉ số phân biệt connection CII. Trên mỗi gói tin nó chứa thông tin về CII để bên nhận dựa vào CII để nhận bộ tham số an toàn tương ứng cho gói tin:

```
case IPPROTO_ESP:
```

```
espp = (struct esp *)(dat + iphlen);
espp->esp_cii = tdbp->tdb_cii;
...
```

Để thêm chẵn khối mã hoá đối với gói tin thì chúng ta đã xác định được độ dài cần thêm vào. Dữ liệu được thêm vào sẽ là 1, 2, 3 . . . Để sang bên nhận khôi phục lại độ dài nguyên bản ban đầu của gói tin thì bên gửi sẽ phải báo độ dài phần dữ liệu thêm vào:
Đoạn trình thực hiện việc thêm dữ liệu chẵn khối:

```
padlen = tailroom - 2 - authlen;

for (i = 0; i < padlen; i++) {
      pad[i] = i + 1;
}
```

Lệnh thực hiện việc báo độ dài phần dữ liệu thêm vào:

```
dat[len - authlen - 2] = padlen;
```

Sau khi đã hoàn tất các công việc cần thiết thì trình sẽ thực hiện việc gọi hàm để mã hoá gói tin được gửi đi bằng việc gọi hàm blockcipher_cbc_encrypt():

```
case ESP_BLOCKCIPHER:
      blockcipher_cbc_encrypt(idat, idat, ilen,
           (caddr_t)(tdbp->tdb_key_e),(caddr_t)iv, 1);
      break;
```

Véc tơ khởi điểm của gói tin sau sẽ được lấy bằng cách cập nhật 16 byte cuối cùng của gói tin được mã hoá trước đó:

```
switch(tdbp->tdb_encalg) {
      case ESP_BLOCKCIPHER:
      /* XXX update IV with the last 16 octets of the encryption */
      ((__u32*)(tdbp->tdb_iv))[0] = ((__u32 *)(idat))[(ilen >> 2) - 4];
      ((__u32*)(tdbp->tdb_iv))[1] = ((__u32 *)(idat))[(ilen >> 2) - 3];
      ((__u32*)(tdbp->tdb_iv))[2] =((__u32 *)(idat))[(ilen >> 2) - 2];
      ((__u32*)(tdbp->tdb_iv))[3] = ((__u32 *)(idat))[(ilen >> 2) - 1];
      break;
```

Sau khi mã hoá gói tin xong, chương trình sẽ thực hiện việc băm trên nội dung gói tin đã được mã để tính ra dữ liệu xác thực. Hàm băm mật mã được dùng ở đây là hàm MD5 với mode HMAC, lấy đầu ra 96 bít (12 byte):

```
switch(tdbp->tdb_authalg) {
      case AH_MD5:
      tctx.md5 = ((struct md5_ctx*)(tdbp->tdb_key_a))->ictx;
      MD5Update(&tctx.md5, (caddr_t)espp, len - iphlen - authlen);
      MD5Final(hash, &tctx.md5);
```

```
      tctx.md5 = ((struct md5_ctx*)(tdbp->tdb_key_a))->octx;
      MD5Update(&tctx.md5, hash, AHMD596_ALEN);
      MD5Final(hash, &tctx.md5);
```

Dữ liệu xác thực được đặt vào ở cuối gói tin sử dụng hàm memcpy():

```
memcpy(&(dat[len - authlen]), hash, authlen);
```

Trong chế độ tunnel, gói tin sau khi đã được mã hoá sẽ được bọc thêm một phần IP header mới với địa chỉ nguồn là địa chỉ IP của card mạng ipsec của nó và địa chỉ đích là địa chỉ IP của card mạng ipsec của máy gateway bảo mật bên nhận. Đoạn trình thực hiện việc gắn thêm địa chỉ IP:

```
case IPPROTO_IPIP:
      iph->version  = 4; /* const should be used here */
      iph->tos      = skb->nh.iph->tos; /* XXX is this right? */
      iph->ttl      = 64; /* ip_statistics.IpDefaultTTL; */
      iph->frag_off = 0;
      iph->saddr    = ((struct sockaddr_in*)(tdbp->tdb_addr_s))-
>sin_addr.s_addr;
      iph->daddr    = ((struct sockaddr_in*)(tdbp->tdb_addr_d))-
>sin_addr.s_addr;
      iph->protocol = IPPROTO_IPIP;
      iph->ihl      = sizeof(struct iphdr) >> 2 /* 5 */;
      iph->id       = htons(ip_id_count++);   /* Race condition here?*/
      newdst = (__u32)iph->daddr;
      newsrc = (__u32)iph->saddr;
      skb->h.ipiph = skb->nh.iph;
      break;
```

Sau đó trình sẽ thực hiện việc xác định đường đi cho gói tin bằng cách gọi hàm ip_route_output():

```
if(ip_route_output(&rt, skb->nh.iph->daddr,  skb->nh.iph->saddr,
            RT_TOS(skb->nh.iph->tos),physdev->iflink)) {
      stats->tx_errors++;
      goto cleanup;
}
```

Gói tin thực sự được gửi đi bằng hàm ip_send():

```
ip_send(skb);
stats->tx_packets++;
```

6. Nhận gói tin trong IP-Crypto

Trong Linux để đăng ký một giao thức mạng nào đó chúng ta sử dụng hàm inet_add_protocol(). Khi gọi hàm này thì trong nhân Linux được thêm vào cấu trúc dùng để xử lý loại gói tin có giao thức mạng cụ thể nào đó. Trong trình của

chúng ta gói tin được bảo mật đó chính là gói tin có giao thức là IPPROTO_ESP. Những gói tin bảo mật đã được hàm ipsec_tunnel_start_xmit( ) xử lý và gán cho giao thức gói tin là IPPROTO_ESP. Và ở bên nhận chỉ có những gói tin nào có giao thức là IPPROTO_ESP mới được trình ipsec xử lý.

Trong hàm ipsec_rcv.c cấu trúc struct inet_protocol esp được xác định như sau:

```
struct inet_protocol esp_protocol =
{
      ipsec_rcv,              /* ESP handler          */
      NULL,                   /* TUNNEL error control */
      0,                      /* next */
      IPPROTO_ESP,                  /* protocol ID */
      0,                      /* copy */
      NULL,                   /* data */
      "ESP"                   /* name */
};
```

Và trong hàm ipsec_init() trong tệp ipsec_init.c thực hiện việc đăng ký giao thức bằng cách gọi hàm:

```
inet_add_protocol(&esp_protocol);
```

Trong cấu trúc esp trên thì ipsec_rcv là hàm dùng để nhận và xử lý gói tin, và IPPROTO_ESP chính là giao thức của gói tin. Như vậy, với các gói tin bảo mật của chúng ta thì giao thức của gói tin bao giờ cũng là IPPROTO_ESP. Và gói tin có giao thức IPPROTO_ESP sẽ được hàm ipsec_rcv() nhận và xử lý. Việc quan trọng nhất của chúng ta đó chính là việc lựa chọn cách xử lý gói tin phù hợp nhất trong hàm ipsec_rcv().

Hàm ipsec_rcv(struct sk_buff *skb, unsigned short xlen) được dùng để xử lý gói tin IPPROT_ESP nhận về. Các công việc cần thực hiện trong hàm ipsec_rcv() đối với gói tin bảo mật đó chính là việc xác thực gói tin, tìm kiếm bộ tham số tương ứng với gói tin, giải mã, loại bỏ phần dữ liệu chẵn khối, loại bỏ phần IP header thêm vào (nếu trong trường hợp tunnel mode) . . . và chuyển gói tin cho tầng mạng cao hơn để xử lý. Trong hàm ipsec_rcv (), bởi vì bộ tham số an toàn cho connection cố định là hàm Mã khối MK1 và hàm băm MD5 nên trong chương trình chúng ta sẽ lựa chọn 2 hàm này để tăng tốc về thời gian.

Việc đầu tiên trong hàm ipsec_rcv() thực hiện chính là việc xác định giao thức của gói tin. Nếu gói tin có giao thức không phải là IPPROTO_ESP thì chương trình sẽ bỏ qua không xử lý gói tin này:

```
protoc = ((struct iphdr *)skb->data)->protocol;
if(protoc != IPPROTO_ESP)  {
      . . .
      goto rcvleave;
```

```
}
```

Sau khi xác định được gói tin có giao thức là IPPROTO_ESP thì chương trình thực hiện việc lấy bộ tham số an toàn ứng với gói tin thông qua các tham số từ gói tin mang tải như cii, địa chỉ nguồn của gói tin và giao thức. Việc chứa đựng tham số cii là rất quan trọng trong gói tin bảo mật. Bởi vì, trên một cặp máy gateway có thể dùng để bảo vệ nhiều cặp mạng con ảo khác nhau. Mà do yêu cầu thực tế mỗi mạng con lại có bộ tham số an toàn khác nhau. Vì thế, đối với gói tin đến trong trường hợp tunnel thì không có cách gì khác hơn là chúng ta phải dựa vào chỉ số cii để biết được gói tin cần phải được dùng bộ tham số an toàn cho connection cụ thể nào trong các bộ tham số an toàn cho các connection hiện tại đang có trên máy gateway.

```
cnid.dst.s_addr = ipp->daddr; /* Lấy địa chỉ nguồn của gói tin */
switch(proto) {
      case IPPROTO_ESP:
            espp = (struct esp *)(skb->data + iphlen);
            cnid.cii = espp->esp_cii; /* Lấy thông số cii của gói tin*/
      break;

ciid.proto = proto; /* Xác định giao thức tương ứng */
```

Hàm lấy bộ tham số an toàn tương ứng với connection:

```
spin_lock_irqsave(&tdb_lock, tdb_flags);
tdbp = gettdb(&cnid);
```

Nếu không tìm thấy bộ tham số an toàn cho gói tin thì gói tin sẽ bị bỏ qua không xử lý:

```
if (tdbp == NULL) {
      . . .
      if(stats) {
            stats->rx_dropped++;
      }
      goto rcvleave;
}
```

Khi tìm thấy bộ tham số an toàn tương ứng với connection hiện tại. Chương trình sẽ thực hiện việc kiểm tra chuỗi số của gói tin xem liệu gói tin đó có bị gửi lặp lại hay là đã quá cũ hay không nhờ hàm ipsec_checkreplaywindow(), nếu gói tin bị lặp hoặc đã quá cũ thì gói tin bị bỏ qua:

```
if (!ipsec_checkreplaywindow(tdbp, replay)) {
      . . .
      if(stats) {
            stats->rx_dropped++;
      }
      tdbp->tdb_replaywin_errs += 1; /* Lỗi xảy ra */
```

```
      goto rcvleave;
}
```

Chương trình sẽ thực hiện tính toán dữ liệu xác thực trên gói tin. Nếu phần dữ liệu xác thực tính được mà không so khớp với phần dữ liệu xác thực mà gói tin mang tải điều đó có nghĩa là nội dung gói tin đã bị thay đổi. Gói tin sẽ không được xử lý: Lệnh lấy dữ liệu xác thực mang tải trên gói tin:

```
authenticator = &(dat[len - authlen]);
```

Đoạn trình tính toán dữ liệu xác thực trên gói tin:

```
if(tdbp->tdb_authalg) {
      switch(tdbp->tdb_authalg) {
      case AH_MD5:
            tctx.md5 = ((struct md5_ctx*)(tdbp->tdb_key_a))->ictx;
            if(proto == IPPROTO_ESP) {
                  MD5Update(&tctx.md5, (caddr_t)espp, ilen);
            }
            MD5Final(hash, &tctx.md5);
            tctx.md5 = ((struct md5_ctx*)(tdbp->tdb_key_a))->octx;
            MD5Update(&tctx.md5, hash, AHMD596_ALEN);
            MD5Final(hash, &tctx.md5);
            break;
      }
```

Việc so sánh dữ liệu xác thực mang tải và dữ liệu xác thực tính được được thực hiện nhờ hàm so sánh memcmp():

```
if (memcmp(hash, authenticator, authlen)) {
      if(stats) {
            stats->rx_dropped++;
      }
      tdbp->tdb_auth_errs += 1;
      goto rcvleave;
} else {
KNMS_PRINT(debug_rcv, "knms_debug:ipsec_rcv: authentication
successful.\n");
}
```

Hàm ipsec_rcv() sau khi đã kiểm tra tính trung thực của nội dung gói tin mang tải, sẽ thực hiện việc giải mã để lấy lại nội dung gói tin nguyên bản ban đầu bằng việc gọi hàm blockcipher_cbc_encrypt():

```
blockcipher_cbc_encrypt(idat, idat, ilen,
                        tdbp->tdb_key_e,(caddr_t)iv, 0);
```

Ở bên gửi đã thực hiện việc thêm dữ liệu cho chẵn khối mã hoá. Thì ở bên nhận chúng ta cần phải loại bỏ phần dữ liệu thêm vào này. Lệnh xác định độ dài phần dữ liệu thêm vào:

```
padlen = idat[ilen - 2];
```

Chúng ta thực hiện việc kiểm tra xem dữ liệu được thêm vào có phải theo thứ tự tuần tự 1, 2, 3, ... hay không? Nếu không phải thì đã có lỗi xảy ra và gói tin bị bỏ qua:

```
for (i = 1; i <= padlen; i++) {
      . . .
      if(i != idat[ilen - 2 - padlen + i - 1]) {
            . . .
            tdbp->tdb_encpad_errs += 1;
      }
}
```

Bởi vì gói tin nguyên dạng ban đầu không có phần ESP header nên khi chúng ta thêm phần ESP header ở đầu gói tin ở bên gửi thì ở bên nhận cũng phải thực hiện loại bỏ phần này đi. Trong Linux việc loại bỏ khoảng dữ liệu ở phần đầu gói tin sẽ được thực hiện với hàm skb_pull():

```
skb_pull(skb, esphlen);
```

Phần dữ liệu thêm vào ở cuối gói tin sẽ được loại bỏ bằng hàm skb_trim():

```
skb_trim(skb, len - esphlen - pad);
```

Trong trường hợp giữa hai gateway bảo mật có thiết lập đường hầm (tunnel) thì phần IP header thêm vào cũng sẽ được loại bỏ:

```
if(ipp->protocol == IPPROTO_IPIP) {
      if(skb->len < iphlen) {
            . . .
            goto rcvleave;
      }
skb_pull(skb, iphlen); /* Hàm loại bỏ phần IP header */
ipp = (struct iphdr *)skb->nh.raw = skb->data;
skb->h.raw = skb->nh.raw + (skb->nh.iph->ihl << 2);
```

Sau khi khôi phục lại dạng gói tin nguyên bản ban đầu thì hàm ipsec_rcv() sẽ thực hiện việc gọi hàm netif_rx() để chuyển phần buffer dữ liệu lên cho tầng giao thức trên xử lý:

```
netif_rx(skb);
```

Gói tin sẽ được đưa lên ở tầng IP, sau đó sẽ được nhân Linux tìm kiếm đường đi tiếp theo của gói tin trong bảng dẫn đường (routing table) và sau đó gói tin được

gửi tới đích (Quá trình này gói tin được đưa về dạng nguyên bản ban đầu, không có mã hoá).

Chương 2
Phần mềm IP-CRYPTO

1. Mã nguồn của IP-CRYPTO

Các file trong thư mục *cryplink-3.0*:

TT	Thư mục cryplink-3.0	Mô tả
1	***Thư mục Cryplink3.0***	
1.1	Makefile	
1.2	version.c	
2	***Thư mục knms/net/ipsec***	Các file chính tạo ra ipsec.o
2.1	knms/net/ipsec/version.c	
2.2	knms/net/ipsec/Config.in	
2.3	knms/net/ipsec/Makefile	
2.4	knms/net/ipsec/defconfig	
2.5	knms/net/ipsec/ipsec_encap.h	
2.6	knms/net/ipsec/ipsec_esp.h	
2.7	knms/net/ipsec/ipsec_init.c	
2.8	knms/net/ipsec/ipsec_ipe4.h	
2.9	knms/net/ipsec/ipsec_md5c.c	
2.10	knms/net/ipsec/ipsec_md5h.h	
2.11	knms/net/ipsec/ipsec_netlink.c	
2.12	knms/net/ipsec/ipsec_netlink.h	
2.13	knms/net/ipsec/ipsec_radij.c	
2.14	knms/net/ipsec/ipsec_radij.h	
2.15	knms/net/ipsec/ipsec_rcv.c	
2.16	knms/net/ipsec/ipsec_rcv.h	
2.17	knms/net/ipsec/ipsec_tunnel.c	
2.18	knms/net/ipsec/ipsec_tunnel.h	
2.19	knms/net/ipsec/ipsec_xform.c	
2.20	knms/net/ipsec/ipsec_xform.h	
2.21	knms/net/ipsec/pfkey_v2.c	
2.22	knms/net/ipsec/pfkey_v2_parser.c	
2.23	knms/net/ipsec/radij.c	
2.24	knms/net/ipsec/radij.h	
2.25	knms/net/ipsec/sysctl_net_ipsec.c	
2.26	knms/net/ipsec/libblockcipher	
3	***Thư mục knms/patches2.2***	Thêm vào mã nguồn của nhân để hỗ trợ IPsec
3.1	knms/patches2.2/Documentation.Configure.help	

3.2	knms/patches2.2/include.linux.in.h	
3.3	knms/patches2.2/include.linux.netlink.h	
3.4	knms/patches2.2/net.Config.in	
3.5	knms/patches2.2/net.Makefile	
3.6	knms/patches2.2/net.ipv4.af_inet.c	
3.7	knms/patches2.2/net.netlink.af_netlink.c	
3.8	knms/patches2.2/net.netlink.netlink_dev.c	
3.9	knms/patches2.2/net.netsyms.c	
3.10	knms/patches2.2/net.netsyms.c.1	
3.11	knms/patches2.2/net.netsyms.c.2	
4	***Thư mục knms/utils***	Các tiện ích phục vụ việc trao đổi giữa người dùng và nhân
4.1	knms/utils/version.c	
4.2	knms/utils/Makefile	
4.3	knms/utils/eroute.c	Tạo ra bảng dẫn đường cho các gói tin IPSEC.
4.4	knms/utils/knmsdebug.c	
4.5	knms/utils/cii.c	Có chức năng tạo và xóa IPSEC CN. Đầu vào là các thông tin trong ipsec.conf sẽ được sử lý để tạo ra các message gửi cho nhân.
4.6	knms/utils/ciigrp.c	Nhóm hoặc hủy nhóm các CN.
4.7	knms/utils/tncfg.c	Gắn card mạng ảo IPSEC với card mạng vật lý.
5	***Thư mục lib***	Các hàm chuyển đổi địa chỉ
5.1	lib/Makefile	
5.2	lib/Makefile.kernel	
5.3	lib/convert.c	
5.4	lib/convert4kernel.c	
5.5	lib/internal.h	
5.6	lib/optionsfrom.c	
5.7	lib/pfkey.h	
5.8	lib/pfkey_v2_build.c	
5.9	lib/pfkey_v2_ext_bits.c	
5.10	lib/pfkey_v2_parse.c	
5.11	lib/pfkeyv2.h	
5.12	lib/cryplink.h	
6	***Thư mục utils***	Các tiện ích chung
6.1	utils/rsasigkey.c	Sử dụng khi dùng phương pháp xác thực

		bằng chữ ký số RSA (tạo ra n, e, d). Chú ý rằng các tham số chỉ được kiểm tra theo thuật toán xác suất.
6.2	utils/Makefile	
6.3	utils/_confread	Thực hiện việc đọc nội dung hai file cấu hình ipsec.conf làm đầu vào cho chương trình cii.
6.4	utils/patcher	Sử dụng khi patch một số file vào nhân để hỗ trợ IPSEC.
6.5	utils/_updown	Thực hiện các lệnh shell như route add, route del, ipfwadm... Sau khi chương trình đọc file ipsec.conf, thông tin sẽ được lưu vào các biến cho _updown sử lý.
6.6	utils/barf	Dùng khi muốn xem các thông tin liên quan khi IPSEC đã chạy.
6.7	utils/errcheck	
6.8	utils/ipsec.in	Tạo ra tệp ipsec sau khi chạy lệnh make. Tệp ipsec có chức năng đơn giản là gọi các lệnh khác để chạy. Chẳng hạn lệnh "ipsec setup start" có nghĩa là thực hiện lệnh "setup start".
6.9	utils/look	Dùng khi muốn xem các thông tin liên quan khi IPSEC đã chạy.
6.10	utils/manual	Được sử dụng khi phân phối khóa theo phương pháp thủ công. Gọi hàm cii để xử lý thông tin đọc từ file ipsec.conf.
6.11	utils/setup	Thực hiện các lệnh để khởi động hoặc tắt dịch vụ ipsec.
6.12	utils/ipsec	
6.13	utils/showdefaults	
7	***Thư mục libblockcipher***	Thuật toán mã hoá
7.1	libblockcipher/Makefile	
7.2	libblockcipher/libblockcipher.a	
7.3	libblockcipher/blockcipher.h	
7.4	libblockcipher/blockcipherconst.h	
8	**Thư mục Keyexchange-3.0**	**Trao đổi khoá tự động**
8.1	**/KeyExchange-3.0/keyingd**	
8.1.1	/keyingd/keyingd.c	
8.2	**KeyExchange-3.0/libgmp**	Chứa thư viện tính toán số lớn gmp
8.3	KeyExchange-3.0/pubkey	Thực hiện trình trao đổi khoá
8.3.1	pubkey/Makefile	
8.3.2	pubkey/Config.h	
8.3.3	pubkey/dhkey.h	Chứa các khai báo cần thiết cho lược đồ

		trao đổi khoá Diffie-Hellman.
8.3.4	pubkey/kex.h	Chứa các khai báo cho trình trao đổi khoá
8.3.5	pubkey/lock.c	Chứa các hàm thực hiện việc khoá file
8.3.6	pubkey/main.c	Chứa hàm main() của trình trao đổi khoá.
8.3.7	pubkey/mk1.h	Chứa các khai báo cho hàm mã hoá
8.3.8	pubkey/negotiate.c	Thực hiện việc trao đổi gói tin
8.3.9	pubkey/p_sha1.c	Chứa hàm băm xác thực sha1
8.3.10	pubkey/packet.c	Chứa các hàm khởi tạo gói tin
8.3.11	pubkey/proto.c	Chứa các hàm xử lý từng kiểu gói tin
8.4	**KeyExchange-3.0/lib**	**Thư mục chứa các trình chuyển đổi**
8.4.1	lib/Makefile	
8.4.2	lib/Makefile.in	
8.4.3	lib/debug.c	
8.4.4	lib/dsprintf.c	
8.4.5	lib/getaddr.c	
8.4.6	lib/gethex.c	
8.4.7	lib/hex.c	
8.4.8	lib/hexdump.c	
8.4.9	lib/hexstr.c	
8.4.10	lib/kex_syslog.c	
8.4.11	lib/kexlib.h	
8.4.12	lib/parseopt.c	
8.4.13	lib/retstatus.c	
8.4.14	lib/secchk.c	
8.4.15	lib/setsig.c	
8.4.16	lib/sighand.c	
8.4.17	lib/xread.c	
8.4.18	lib/xwrite.c	
8.4.19	lib/xwritev.c	
8.5	**KeyExchange-3.0/libblockcipher**	**Thư mục chứa thư viện của hàm mã khối**

2. Quá trình biên dịch và cài đặt IP-CRYPTO

2.1 Các yêu cầu

- Máy tính cài đặt bản Linux Red Hat 6.2 (phiên bản nhân 2.2.14-5.0).
- Thư mục chứa mã nguồn (file **cryplink-3.0.tar.gz**) là /usr/src.
- Sử dụng phiên làm việc với người dùng root (siêu người dùng).
- Phải đảm bảo đã cài đặt gói kernel-source-2.2.14-5.0.i386.rpm và gói kernel-headers-2.2.14-5.0.i386.rpm. Ngoài ra các gói phục vụ quá trình biên dịch cũng phải được cài đặt.

- Người biên dịch phải biết cách cài đặt và chạy thử nhân mới không có hỗ trợ CrypLink trước khi thực hiện quá trình này.

Sau khi copy file mã nguồn **cryplink-3.0.tar.fz** vào thư mục /usr/src chúng ta phải giải nén file này:

[root@pvkh /]# cp cryplink-3.0.tar.gz /usr/src/
[root@pvkh /]# tar -xzvf /usr/src/cryplink-3.0.tar.gz

Sau khi giải nén chúng ta sẽ có thư mục cryplink-1.0 trong /usr/src. Ở đây, chúng tôi sẽ trình bày hai phương pháp biên dịch để cài đặt phần mềm CrypLink. Phương pháp thứ nhất là cài đặt ở chế độ kernel. Toàn bộ KNMS được gắn trực tiếp vào trong mã nguồn của nhân. Phương pháp thứ hai là cài đặt ở chế độ module. KNMS được dịch ở dạng module và được gắn vào nhân khi CrypLink khởi động.

2.2 Cài đặt ở chế độ kernel

- **Bước 1:**

Chuyển vào thư mục cryplink-3.0 sau khi đã giải nén. Sau đó thực hiện các lệnh sau:

[root@pvkh /]# cd /usr/src/cryplink-3.0
[root@pvkh /cryplink-3.0]# make insert
[root@pvkh /cryplink-3.0]# make programs
[root@pvkh /cryplink-3.0]# make install

Lệnh **make insert** thực hiện việc patch các file cần thiết vào mã nguồn của nhân để nhân hỗ trợ IPSEC; tạo ra thư mục ipsec trong /usr/src/linux-2.2.14-5.0/net/ có các file và thư mục liên kết đến các file mã nguồn của cryplink. Lần đầu tiên, lệnh này cũng tạo ra file /dev/ipsec để trao đổi với KNMS.

- **Bước 2:**

Cấu hình và cài đặt nhân mới có hỗ trợ CrypLink. Chúng ta phải vào thư mục mã nguồn của nhân /usr/src/linux-2.2.14-5.0 chạy các lệnh sau đây để cấu hình và cài đặt nhân mới có hỗ trợ CrypLink:

[root@pvkh /cryplink-3.0] cd /usr/src/linux
[root@pvkh /linux] make menuconfig

Lệnh **make menuconfig** sẽ đưa ra một danh sách các menu để lựa chọn các tham số cấu hình nhân. Ta chỉ quan tâm đến tuỳ chọn **Networking options**. Các tham số cho CrypLink bao gồm:

- CONFIG_IPSEC: Nếu chọn ‘y’ thì sẽ được link tĩnh vào kernel, nếu chọn ‘m’ thì có thể đưa vào nhân ở dạng module, chọn ‘n’ thì sẽ bỏ tất cả.

- CONFIG_IPSEC_PFKEYv2: PF_KEYv2 kernel/user interface

Tham số này cung cấp cho RFC2367 định nghĩa ra PF_KEYv2 sockets đến kernel để gửi thông tin về khóa (từ keying daemon là keyingd).

- CONFIG_IPSEC_IPIP: IP-in-IP encapsulation

  Hỗ trợ chế độ đường ngầm (tunnel). Nếu bạn thiết lập IPSEC để bảo vệ thông tin giữa hai mạng con qua hai gateway thì lựa chọn này phải được thiết lập.

- CONFIG_IPSEC_ICMP: Enable ICMP PMTU messages

- CONFIG_IPSEC_ESP: Encapsulating Security Payload

  Tuỳ chọn này cung cấp việc bảo mật và xác thực nội dung gói tin.

- CONFIG_AUTH_HMAC_MD5

  Tuỳ chọn này cung cấp việc xác thực gói tin theo phương pháp HMAC_MD5.

- CONFIG_IPSEC_ENC_BLICKCIPHER

  Tuỳ chọn này cung cấp việc mã hoá gói tin theo thuật toán mã khối MK1 hoặc GOST.

- DEBUG_IPSEC: IPSEC Debugging Option

  Thiết lập thông tin gỡ rối khi chạy IPSEC.

Sau khi đã ghi lại cấu hình vừa thay đổi, chạy các lệnh sau để tạo và cài đặt nhân mới.

[root@pvkh /linux] make dep; make clean; make install

hoặc:

[root@pvkh /linux] make dep; make clean; make bzImage
[root@pvkh /linux] copy arch/i386/boot/bzImage /boot/kernel-cryplink

Thêm các dòng sau vào cuối file /etc/lilo.conf:

image = /boot/kernel-cryplink
label = cryplink
read-only
root = /dev/hda1

Chạy lệnh:

[root@pvkh /linux] lilo

Đến đây, quá trình biên dịch và cài đặt đã hoàn thành. Nếu khởi động lại máy thì chúng ta sẽ có một nhân mới hỗ trợ CrypLink. Nhưng để chạy được thì phải thiết lập cấu hình cho hai file ipsec.conf và ipsec.private của CrypLink. Phần này được trình bày ở mục 4.2.

2.3 Cài đặt ở chế độ module

- **Bước 1:**

  Giống **Bước 1** ở chế độ kernel.

- **Bước 2:**

  [root@pvkh /cryplink-3.0] cd /usr/src/linux
  [root@pvkh /linux] make menuconfig

  Đánh dấu **M** vào tuỳ chọn **IP Security Protocol (CrypLink IPSEC)** trong menu **Networking options**. **M** có nghĩa là biên dịch phần CrypLink trong nhân thành một module. Khi CrypLink khởi động, nhân sẽ tự động nạp module này. Các tham số sau được lựa chọn tương tự.

  Sau khi ghi lại cấu hình, thực hiện các bước sau để tạo module **ipsec.o**:
  [root@pvkh /linux] make dep; make clean
  [root@pvkh /linux] make modules; make modules_install

Sau bước **make modules**, nếu thành công, chúng ta tạo được module ipsec.o trong thư mục /usr/src/linux/net/ipsec. Lệnh **make modules_install** sẽ cài đặt module này vào thư mục /lib/modules/2.2.14-5.0/misc để nhân nạp khi CrypLink khởi động. Lệnh **make modules_install** có thể không cần thiết. Thay lệnh này bằng việc copy file ipsec.o tạo được vào thư mục /lib/modules/2.2.14-5.0/misc.

Đến đây quá trình biên dịch và cài đặt đã hoàn thành. Có một điều cần chú ý là chúng ta vẫn sử dụng nhân cũ và nó phải được dịch mà không có phần hỗ trợ CrypLink hoặc một phần mềm tương tự.

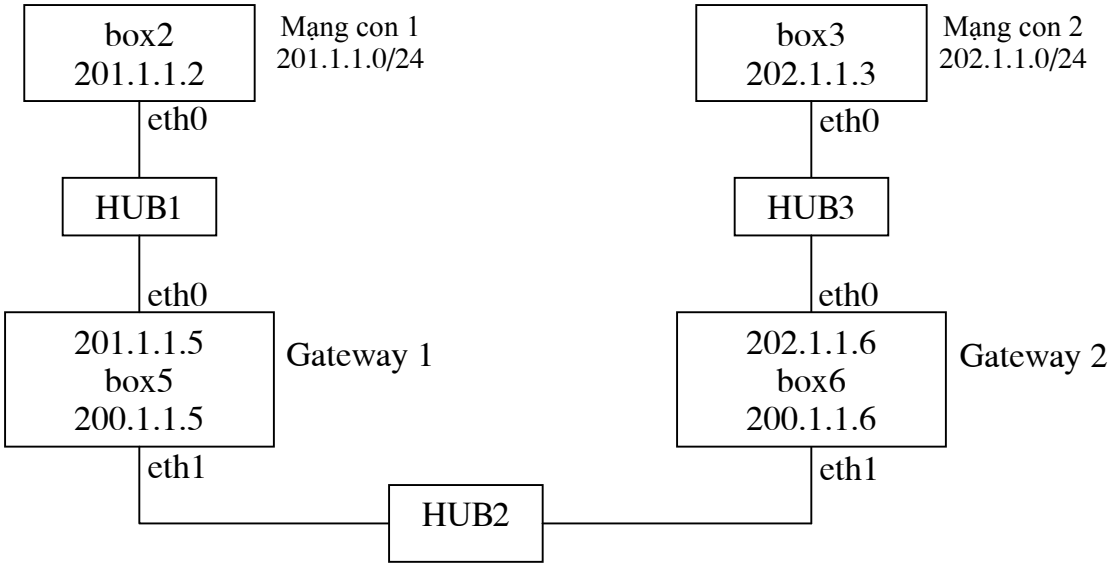
Nhận xét

Quá trình biên dịch ở chế độ module tiện lợi hơn khi dịch ở chế độ kernel. Bởi vì ở chế độ kernel, chúng ta phải dịch và cài đặt lại nhân mới mỗi khi cài đặt CrypLink. Trong chế độ module chúng tôi đã thay đổi Makefile để quá trình biên dịch module chỉ dịch riêng một module ipsec.o, khi đó quá trình biên dịch và cài đặt CrypLink sẽ nhanh lên rất nhiều.

3. Thiết lập cấu hình cho IP-CRYPTO

3.1 Cấu hình mạng

Chúng tôi lấy mô hình mạng tại phòng thí nghiệm an toàn mạng máy tính của PVKH MM làm ví dụ. Tuỳ thuộc vào mô hình mạng của bạn mà khi cấu hình sẽ có những thay đổi.



Theo mô hình trên: box5, box6 là hai gateway được nối với nhau qua HUB2. box2, box3 lần lượt là một máy trong mạng con của box5, box6. Chúng ta sẽ thiết lập IPSEC trên hai gateway box5, box6 để bảo vệ thông tin trao đổi giữa hai mạng con 201.1.1.0/24 và 202.1.1.0/24.

Sau khi đã thiết lập mạng theo mô hình như trên, chúng ta cần thực hiện các bước sau:

- Chạy **linuxconf** hoặc **ifconfig** để thiết lập gateway mặc định cho box5 là box6 (200.1.1.6) và ngược lại.
- Đặt tuỳ chọn net.ipv4.ip_forward = 1 trong file /etc/sysctl.conf hoặc sử dụng lệnh: **echo 1 > /proc/sys/net/ipv4/ip_forward**
- Khởi động lại dịch vụ mạng: **/etc/rc.d/init.d/network restart**
- Thử dùng lệnh ping từ máy box2 sang box3, nếu ping được có nghĩa là cấu hình mạng thành công.

3.2 Cấu hình IP-CRYPTO

CrypLink cho phép thiết lập kết nối giữa hai mạng con theo hai chế độ: trao đổi khoá thủ công và trao đổi khoá tự động.

Trao đổi khoá thủ công

Ở chế độ này khoá phải được tạo trước và ghi vào trong file cấu hình (/etc/ipsec.conf). Chế độ này kém an toàn hơn so với chế độ trao đổi khoá tự động.

- **Tại box5:**

Tạo file /etc/ipsec.conf (**touch /etc/ipsec.conf**) có dạng:

```
# File cấu hình CrypLink -  /etc/ipsec.conf
config setup
        interfaces = "ipsec0 =  eth1"
        knmsdebug = none
        plutodebug = none
        plutoload = %search
        plutostart = %search
conn box5_box6
        keyingtries = 0
        cii = 0x200
        esp = True
        # khoá mã dịch có độ dài 512 bit (ở đây lấy ví dụ là 16 số dạng
        # 0x12345678 cách nhau bởi dấu gạch dưới)
        espenckey = [0x12345678_12345678_..._12345678]
        # khoá xác thực có độ dài 128 bit
         authkey = 0x12345678_12345678_12345678_12345678
        left = 200.1.1.5
        leftsubnet = 201.1.1.0/24
        right = 200.1.1.6
        #box5_box6
        rightsubnet = 202.1.1.10/24
```

Ý nghĩa của các tham số trên như sau:

- **interfaces = "ipsec0 = eth1"**: tham số này chỉ ra card mạng ảo ipsec0 được gắn vào card mạng vật lý eth1 khi IPSEC hoạt động.
- cii: số CII được sử dụng cho sự kết nối. Số này phải ở dạng 0x*hex*, ở đây *hex* là một hoặc nhiều số thập lục phân (hexadecimal) (chú ý, nhìn chung cần tạo một số cii nhỏ nhất là 0x100 để KNMS có thể chấp nhận được).
- esp: xác định gói tin được sử dụng ở đây là gói tin ESP.
- espenckey: khóa mã hóa của ESP.
- espauthkey: khóa xác thực của ESP.
- left: tham số này xác định địa chỉ IP của cổng an ninh bên trái (left participant) của giao diện mạng công cộng.
- leftsubnet: tham số này xác định mạng con cổng an ninh bên trái, được biểu diễn như là network/netmask. Nếu tham số này bị bỏ qua thì giả định là left/32, có nghĩa rằng đầu cuối bên trái của sự kết nối chỉ là cổng an ninh bên trái.

- **Tại box6:**

  Tạo file ipsec.conf trong thư mục /etc giống như ở box5.

<u>Trao đổi khoá tự động:</u>

- Yêu cầu: Máy tính chạy hệ điều hành Linux RedHat 6.2 (nhân 2.2.14) hoặc RedHat 7.0 (nhân 2.2.16). Người sử dụng phải là root. Giả sử tên hai máy tính lần lượt là box5 và box6. Để biên dịch được chương trình nguồn, chúng ta cần bộ nguồn OpenSSL phiên bản 0.9.6 hoặc mới hơn.
- Tạo thư mục /etc/keyEx/pk, /etc/keyEx/run.

- Tạo hai bộ khoá công khai RSA cho hai bên như sau: (n1, e1, d1) cho box5 và (n2, e2, d2) cho box6.
- **Trên máy box5 tạo hai file:**
  - file thứ nhất: /etc/keyEx/keyEx.priv lưu n1 và d1 như sau:

  # PVKH: file /etc/keyEx/keyEx.priv
  myModulo=0xn1
  PrivateExponent=0xd1

  - file thứ hai: /etc/keyEx/pk/box6 lưu n2 và e2 như sau:

  # PVKH: file /etc/keyEx/pk/box6
  peerModulo=0xn2
  peerPublicExponent=0xe2

- **Trên máy box6 tạo hai file tương tự như trên**
  - file thứ nhất: /etc/keyEx/keyEx.priv lưu n2 và d2 như sau:

  # PVKH: file /etc/keyEx/keyEx.priv
  myModulo=0xn2
  PrivateExponent=0xd2

  - file thứ hai: /etc/keyEx/pk/box5 lưu n1 và e1 như sau:

  # PVKH: file /etc/keyEx/pk/box5
  peerModulo=0xn1
  peerPublicExponent=0xe1

Ví dụ: với cấu hình của chúng tôi, trên máy box5 là:

- file /etc/keyEx/pk/box6

peerModulo=0x89f5d9ec8170d69bfd403d9ed4037ff14a01abfdfb
e94d5ef64b73a32d22cfaf19f1a9b1b403b6d7b0f922c425085c9f3
4af3b64aaf08af16a498d4814c69da3

peerPublicExponent=0x40c14214bfcaba17db7549cf934fe73f12
345d9855f8c9c10d5525f26a76545441537a74cc58867fdaf02d892
96b29c6cfff1b17fdd08eb4abc200b1b408c2130789f5d9ec8170d6
9bfd403d9ed4037ff14a01abfdfbe94d5ef64b73a32d22cfaf19f1a
9b1b403b6d7b0f922c425085c9f34af3b64aaf08af16a498d4814c6
9da3

Chú ý: khoá được ghi trên một dòng, trong file có thể có dấu cách, dấu '#'.

- file /etc/keyEx/keyEx.priv

myModulo=0x95441d0731fd9f5a1dc6dfb2f2eadec98024f8b013ab
dc772d72346b9309cd59d3e68f92dff6e39051a792caac15a01fb58
26f337524c87e1bfedbb954ccef89

PrivateExponent=0x209c36eadf507badba36743d15073bd36503b
d9c2702f3f966befdd8ef2ac10c4cb15ddf7a5880db674bb3b0928b
a5d17d54db04436b234a6f37701e35330edd

- **Trên máy box6:**

  - file thứ nhất: /etc/keyEx/keyEx.priv lưu n2 và d2 như sau:

myModulo=0x89f5d9ec8170d69bfd403d9ed4037ff14a01abfdfbe9
4d5ef64b73a32d22cfaf19f1a9b1b403b6d7b0f922c425085c9f34a
f3b64aaf08af16a498d4814c69da3

PrivateExponent=0x48d1dd23dc94e7ab929cca519caeab6549aa4
f1fd260baeacd9e1705851fa1609a25ec1b161e64a85af98d12b087
b59534832feed40c7f53a815c0e7944ff0ff

  - file thứ hai: /etc/keyEx/pk/box5 lưu n1 và e1 như sau:

peerModulo=0x95441d0731fd9f5a1dc6dfb2f2eadec98024f8b013
abdc772d72346b9309cd59d3e68f92dff6e39051a792caac15a01fb
5826f337524c87e1bfedbb954ccef89

peerPublicExponent=0x40c43eb9f5d50f8c01bba8bf0ee8e6dae3
8b4de9783734d139534bfbd4d5fac2072cea07ae0a8ff566d663561
a45f74d73bb51892458ef940137656e9e2949db4595441d0731fd9f
5a1dc6dfb2f2eadec98024f8b013abdc772d72346b9309cd59d3e68
f92dff6e39051a792caac15a01fb5826f337524c87e1bfedbb954cc
ef89

- copy file chạy `kex` vào thư mục /usr/sbin
- Lựa chọn cổng 777 (phải đảm bảo cổng này chưa có dịch vụ nào sử dụng) cho kex bằng cách thêm dòng sau vào file /etc/services:
  kex 777/tcp #port 777 for kex
- Thêm dòng sau vào file /etc/inetd.conf
  kex stream tcp nowait root /usr/sbin/kex kex
  hoặc nếu sử dụng TCP Wrapper cho việc điều khiển truy nhập:
  kex stream tcp nowait root /usr/sbin/tcpd /usr/sbin/kex

Sau khi cấu hình xong với 2 file /etc/services và /etc/inetd.conf, ta nên khởi động lại máy để kích hoạt chương trình `kex`.

Lệnh chạy `kex`:

Trên box5:

kex -c box6:777

Sau khi thực hiện lệnh này thì khoá tạo ra được lưu trong /etc/keyEx/run/box6

(Chý ý: Nếu chạy trên box6 thì thực hiện lệnh `kex -c box5:777` - khoá tạo ra là file /etc/keyEx/run/box5).

Trong chương trình trao đổi khoá tự động, người thiết lập cấu hình cho từng cặp connection cần phải chú ý một điều sau: Chúng ta cần phải đặt tên connection nằm ở sau dấu # ở ngay sau địa chỉ IP của máy bảo mật bên kia. Ví dụ, ở trên máy box5:

conn box5_box6

```
. . .
left = 200.1.1.5
leftsubnet = 201.1.1.0/24
right = 200.1.1.6
#box5_box6
rightsubnet = 202.1.1.10/24
```

Ở trên máy box6:

conn box5_box6

```
. . .
left = 200.1.1.5
#box5_box6
leftsubnet = 201.1.1.0/24
right = 200.1.1.6
rightsubnet = 202.1.1.10/24
```

Sử dụng trình keyingd

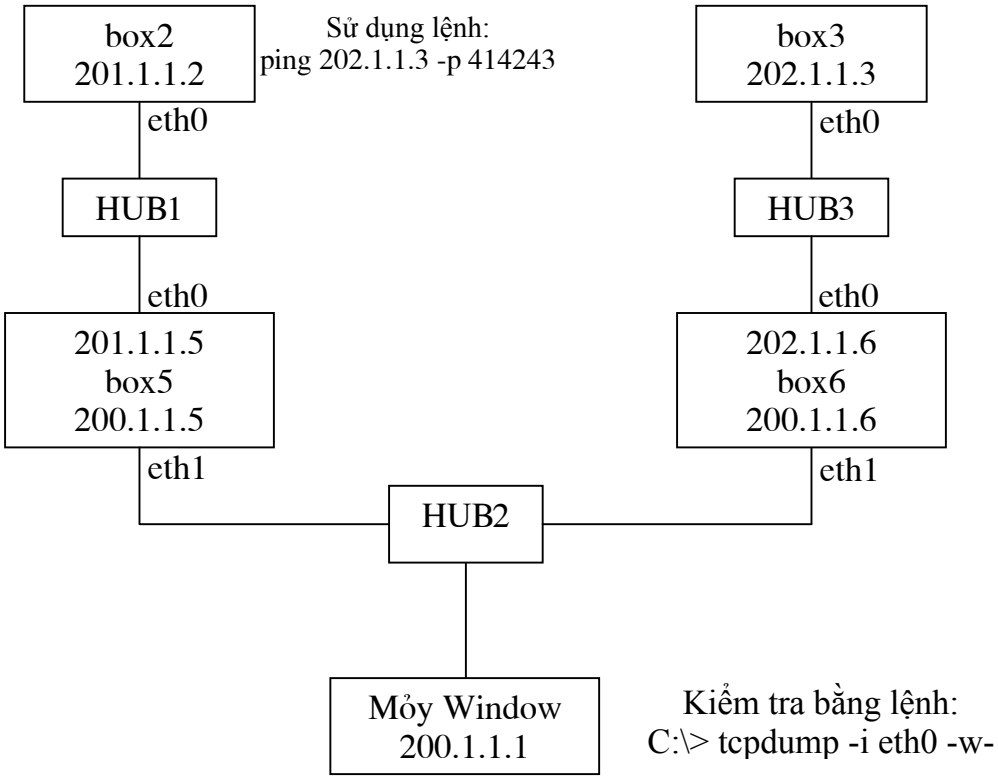
Trong chương trình ipsec, chúng ta có thể dùng chương trình keyingd để thực hiện việc trao đổi khoá lại cho từng cặp connection sau 6 giờ. Để chạy chương trình keyingd bạn chỉ cần gọi keyingd với tham số là tên của connection: Ví dụ, chúng ta cần up connection là box5_box6, trên box5 hoặc box6 chúng ta đều có thể gọi lệnh sau:

keyingd –c box5_box6

Kiểm tra quá trình cài đặt và cấu hình

- Khởi động lại box5 và box6 để khởi động Cryplink (**ipsec setup start**).
- Đối với trường hợp trao đổi khoá thủ công sử dụng lệnh: **ipsec manual --up sample** ở cả hai máy để IPSEC hoạt động. Đối với trường hợp trao đổi khoá tự động sử dụng lệnh: kex –c box5:777

- Tại box2 (201.1.1.2) chạy lệnh **ping 202.1.1.3**, nếu ping được thì quá trình thiết lập cấu hình thành công. Nếu không, phải kiểm tra lại các bước trên .
- Sử dụng các lệnh **ipsec barf**, **ipsec look**, **ipsec tncfg** để xem các thông tin gỡ rối.
- Nếu lệnh ping không chạy hãy kiểm tra giá trị ip_forward trong /proc/sys/net/ipv4 có là 1 không, sau đó kiểm tra lại quá trình cấu hình ipsec.
- Dùng chương trình tcpdump để kiểm tra thông tin trao đổi giữa box5 và box6 đã được mã hoá chưa. Tại box2 dùng lệnh **ping 202.1.1.3 -p 414243**, lệnh ping với tham số -p 414243 sẽ gửi dữ liệu là ba ký tự A (41) B (42) C(43) sang máy 202.1.1.3. Tại máy Window (200.1.1.1) kiểm tra bằng lệnh **tcpdump -i eth0 -w-** Thông tin hiện trên màn hình cho biết có mã hoá hay không.



- Tại **box6** (hoặc box5) sử dụng lệnh **ipsec look** để xem các thông tin liên quan:

```
box6 Tue Jun 26 10:55:17 ICT 2001
============-============
202.1.1.0/24       -> 201.1.1.0/24       => tun0x104@200.1.1.5 esp0x464abd31@200.1.1.5
------------=------------
esp0x464abd30@200.1.1.5 ESP_Ma_Khoi_HMAC_MD5: dir=out ooowin=32 alen=128 aklen=128 eklen=512
life(c,s,h)=add(12766,0,0)
esp0x464abd31@200.1.1.5 ESP_Ma_Khoi_HMAC_MD5: dir=out ooowin=32 alen=128 aklen=128 eklen=512
life(c,s,h)=add(12767,0,0)
esp0xc574a827@200.1.1.6 ESP_Ma_Khoi_HMAC_MD5: dir=in ooowin=32 alen=128 aklen=128 eklen=512
life(c,s,h)=add(12766,0,0)
esp0xc574a828@200.1.1.6 ESP_Ma_Khoi_HMAC_MD5: dir=in ooowin=32 alen=128 aklen=128 eklen=512
life(c,s,h)=add(12767,0,0)
tun0x101@200.1.1.6 IPIP: dir=in 200.1.1.5 -> 200.1.1.6 life(c,s,h)=add(12766,0,0)
tun0x102@200.1.1.5 IPIP: dir=out 200.1.1.6 -> 200.1.1.5 life(c,s,h)=add(12766,0,0)
tun0x103@200.1.1.6 IPIP: dir=in 200.1.1.5 -> 200.1.1.6 life(c,s,h)=add(12767,0,0)
```

```
tun0x104@200.1.1.5 IPIP: dir=out 200.1.1.6 -> 200.1.1.5 life(c,s,h)=add(12767,0,0)
Destination     Gateway         Genmask         Flags   MSS Window  irtt Iface
0.0.0.0         200.1.1.5       0.0.0.0         UG        0 0          0 eth1
200.1.1.0       0.0.0.0         255.255.255.0   U         0 0          0 eth1
200.1.1.0       0.0.0.0         255.255.255.0   U         0 0          0 ipsec0
200.1.1.6       0.0.0.0         255.255.255.255 UH        0 0          0 eth1
201.1.1.0       200.1.1.5       255.255.255.0   UG        0 0          0 ipsec0
```

4. Các tệp sau khi cài đặt

tt	Tệp	Chức năng/cách tạo ra
1	/etc/rc.d/init.d/ipsec	Thực hiện các lệnh để khởi động hoặc tắt dịch vụ ipsec
2	/etc/rc.d/rc0.d/K68ipsec	Cách đánh số xem trong file /etc/rc.d/init.d/ipsec
3	/etc/rd.c/rc1.d/K68ipsec	
4	/etc/rd.c/rc2.d/S47ipsec	
5	/etc/rc.d/rc3.d/S47ipsec	
6	/etc/rd.c/rc4.d/S47ipsec	
7	/etc/rd.c/rc5.d/S47ipsec	
8	/etc/rd.c/rc6.d/K68ipsec	
9	/etc/ipsec.conf (text)	File cấu hình IPsec
10	/usr/lib/ipsec (shell)	Gọi các chương trình (tham số đầu tiên là chương trình cần chạy, tham số thứ hai trở đi là tham số cho chương trình đó)
11	/usr/lib/ipsec/cii (ELF)	Có chức năng tạo và xóa IPSEC CN. Đầu vào là các thông tin trong ipsec.conf sẽ được sử lý để tạo ra các message gửi cho nhân.
12	/usr/lib/ipsec/eroute (ELF)	Tạo ra bảng dẫn đường cho các gói tin IPSEC. Một bảng dẫn đường bao gồm giao thức
13	/usr/lib/ipsec/ciigrp (ELF)	Nhóm hoặc hủy các tổ hợp bảo vệ
14	/usr/lib/ipsec/tncfg (ELF)	Gắn card mạng ảo IPSEC với card mạng vật lý.
15	/usr/lib/ipsec/klipdebug (ELF)	Đưa ra các thông tin gỡ rối KNMS
16	/usr/lib/ipsec/barf, look (shell)	Đưa ra các thông tin liên quan khi IPsec đã chạy
17	/usr/lib/ipsec/manual (shell)	Được sử dụng khi phân phối khóa theo phương pháp thủ công. Gọi hàm cii để xử lý thông tin đọc từ file ipsec.conf.
18	/usr/lib/ipsec/showdefaults (shell)	Hiện thông tin đường dẫn mặc định trong file ipsec.conf nếu được thiết lập (interface=%defaultroute)

19	/usr/lib/ipsec/*_confread	Thực hiện việc đọc nội dung hai file cấu hình ipsec.conf để làm đầu vào cho chương trình (manual).
20	/usr/lib/ipsec/_updown	Thực hiện các lệnh shell như route add, route del, ipfwadm... Sau khi chương trình đọc file ipsec.conf, thông tin sẽ được lưu vào các biến cho _updown sử lý (xem hàm do_command() trong kernel.c).
21	/usr/lib/ipsec/patcher (shell)	Sử dụng khi patch một số file vào nhân.
22	/usr/lib/ipsec/setup (shell)	Đây là file liên kết đến /etc/rc.d/init.d/ ipsec
23	/usr/sbin/ipsec (shell)	giống /usr/lib/ipsec
24	/lib/modules/2.2.14-50/misc	ipsec.o (module)
25	/usr/sbin/kex	Trình thực hiện việc trao đổi khoá tự động
26	/usr/sbin/keyingd	Trình quản lý việc trao đổi khoá tự động về mặt thời gian.

C. PHẦN MỀM DL-CRYPTOR

Chương 1
Bảo mật ở tầng DataLink

Data link là tầng thứ hai trong mô hình 7 tầng của OSI. Ở tầng data link hệ điều hành không biết tới gói tin nhận được thuộc giao thức gì (IP, IPX, AppleTalk ...). Vì vậy việc can thiệp vào tầng data link là một giải pháp khá tổng thể trong việc bảo mật mạng.

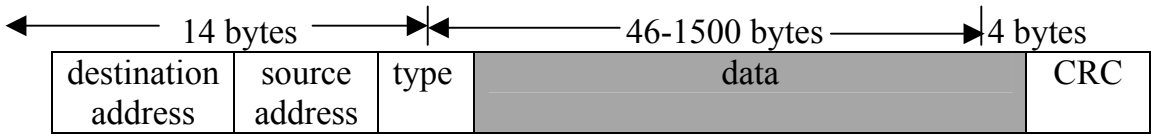
1. Cấu trúc gói tin MAC (Medium Access Control)

Giao thức MAC (Medium Access Control - Điều khiển truy nhập thiết bị) được sử dụng để cung cấp cho tầng data link (liên kết dữ liệu) của hệ thống mạng Ethernet LAN. Giao thức MAC bọc phần dữ liệu SDU bằng cách thêm vào 14 byte header (Protocol Control Information (PCI) – Thông tin điều khiển giao thức) trước phần dữ liệu và thêm 4-byte (32-bít) mã kiểm tra CRC (Cyclic Redundancy Check) sau dữ liệu. Tất cả gói tin được nằm sau 8 byte dữ liệu thêm ở đầu.

- **Preamble**

  Khoảng thời gian ngắn trước khi quá trình truyền dữ liệu bắt đầu cho phép có một khoảng thời gian nhỏ để đầu nhận ở mỗi nút (receiver electronics node) thiết đặt trạng thái sau khi đã nhận hoàn tất gói tin trước đó. Ở nút gửi bắt đầu truyền dữ liệu bằng cách gửi 8 byte (64 bít) chuỗi tuần tự. Nó bao gồm 62 con số 1 và 0 đặt sau mặt nạ 11. Byte cuối cùng mà kết thúc với '11' được biết đến như dấu hiệu 'Start of Frame Delimiter" (Bắt đầu truyền gói tin).

- **Header (Phần đầu)**



Header (phần đầu) bào gồm 3 phần sau:

- 6 byte địa chỉ đích, 6 byte địa chỉ này xác định nút nhận đơn (single) trong gói tin unicast, và nhóm nút nhận trong gói tin multicast, hoặc là tập tất cả các nút nhận trong chế độ broadcast.
- 6 byte nguồn, nó được lập để xác định địa chỉ duy nhất của người gửi. Nó có thể được dùng bởi giao thức tầng mạng (network layer protocol) để định danh người gửi.
- 2 byte trường kiểu (type), nó cung cấp cho SAP (Service Access Point - Điểm truy cập dịch vụ) định danh kiểu giao thức đi kèm (ví dụ, giá trị 0x0800 được sử dụng để xác định giao thức mạng IP, các giá trị khác được sử dụng để chỉ ra các giao thức tầng mạng khác).

- **CRC**

  32 bít CRC được thêm vào ở cuối gói tin để cung cấp thông tin cho việc phát hiện lỗi khi trên đường truyền có lỗi xảy ra (hoặc là lỗi xung đột phát trong mạng Ethernet) kết quả là gây ra lỗi cho gói tin MAC. Mọi gói tin mà có CRC không đúng sẽ bị nơi MAC nhận bỏ qua mà không xử lý tiếp. Giao thức MAC không cung cấp thông tin nào về việc gói tin bị bỏ qua do lỗi CRC.

2. Lập trình module bảo mật mạng

2.1 Lập trình module

Module cũng giống như hầu hết mọi chương trình khác chỉ khác rằng các trình ứng dụng bình thường thì chạy trong môi trường người dùng còn module thì chạy trong nhân của hệ điều hành. Vì thế, chúng phải được định nghĩa MODULE và phải khai báo (include) file header module.h, cùng với các file kernel header cần thiết khác mà định nghĩa hàm hoặc các biến mà được dùng trong module. Module có thể khá đơn giản, nhưng chúng cũng có thể là khá phức tạp như các trình điều khiển thiết bị và hệ thống file.

Sau đây là định dạng chung của một module:

```
#define MODULE
#include <linux/module.h>
/* ... các file header cần thiết khác ... */

/*
* ... các khai báo cho module và các hàm sử dụng trong module ...
*/

int init_module() {
/* mã kernel sẽ gọi khi chúng ta cài đặt module */
...
}

void cleanup_module( ) {
/* mã kernel sẽ gọi khi chúng ta loại bỏ module */
}
```

Chúng ta phải chú ý rằng không phải tất cả mọi biến trong kernel cũng được xuất khẩu (export) cho các module sử dụng, cho dù là trong đoạn trình chúng ta có khai báo extern. Gần đây nhân linux có xuất khẩu cả hai đối tượng (symbol) và phiên bản của nó sử dụng macro EXPORT_SYMBOL(x). Đối với biến mà do người dùng tạo ra, sử dụng macro EXPORT_SYMBOL_NOVERS(x) để thay thế, trình liên kết sẽ không tìm biến này trong bảng đối tượng của nhân. Người lập trình module có thể muốn sử dụng macro EXPORT_NO_SYMBOL, theo ngầm định module sẽ xuất khẩu tất cả các biến của nó.

2.2 Cài đặt và xoá bỏ module

Cài đặt và xoá bỏ module đơn giản như gọi một chương trình với tên của module biên dịch. Bạn phải là siêu người dùng (super user) thì mới được phép cài đặt hoặc loại bỏ module. Chương trình insmod sẽ tiến hành cài đặt module, đầu tiên nó sẽ liên kết module với bảng đối tượng mà nhân xuất khẩu để giải quyết vấn đề về tham chiếu và sau đó nó sẽ cài đặt mã vào trong nhân (kernel space).

/sbin/insmod module_name

Chương trình rmmod sẽ xoá bỏ module đã được cài đặt và mọi tham chiếu mà nó đã xuất khẩu.

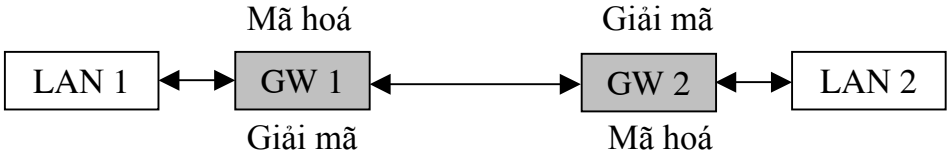
/sbin/rmmod module_name

Trình lsmod sẽ liệt kê danh sách tất cả các module hiện tại được nạp

/sbin/lsmod

2.3 Module bảo mật mạng ở tầng datalink

Trong nhân linux việc gửi và nhận gói tin mạng được chứa trong cấu trúc chứa gói tin struct sk_buff. Mọi xử lý ở các tầng khác nhau đều xử lý trên cấu trúc này. Ta thấy trong nhân linux việc gửi và nhận gói tin ở tầng data link được thực hiện nhờ hai hàm là dev_queue_xmit() trong trường hợp gửi gói tin đi và net_bh() trong trường hợp nhận gói tin. Hàm dev_queue_xmit() sẽ chuyển dữ liệu vào hàng đợi cho giao diện vật lý gửi gói tin đi. Mặt khác hàm net_bh() sẽ lấy gói tin do giao diện vật lý nhận được đưa vào bộ đệm hàng đợi để chuyển lên cho các giao thức ở trên xử lý. Vì vậy chúng ta thấy để can thiệp mật mã vào tầng data link thì giải pháp can thiệp vào hai hàm này là phương pháp tối ưu nhất. Khi gói tin được truyền đi, hàm dev_queue_xmit() sẽ thực hiện việc mã hoá và sang bên nhận hàm net_bh() sẽ thực hiện việc giải mã. Như vậy, đối với các giao thức mạng ở tầng cao hơn (ví dụ, giao thức tầng mạng IP) ở hai máy là trong suốt.



Trong máy thực hiện việc định tuyến (router, gateway) thì việc gói tin đi ra card mạng nào thì được mã là điều rất quan trọng. Bởi vì hàm dev_queue_xmit() và hàm net_bh() được hệ điều hành dùng chung cho mọi giao diện. Nếu tất cả mọi gói tin đi và đến đều thực hiện mã hoá hoặc giải mã thì hệ thống của chúng ta có thể không làm việc được nữa (bởi vì gói tin được mã hoá hay giải mã một chiều thì cấu trúc của nó bị phá vỡ và các giao thức ở tầng trên không thể xử lý được nữa). Rất may, trong linux việc lựa chọn gói tin đi ra hay nhận về card mạng nào thì được mã hoá hay giải mã là bài toán rất đơn giản. Bởi vì, gói tin nhận hay gửi trong cấu trúc sk_buff của gói tin đều có chứa cấu trúc thiết bị (struct device) mà gửi hay nhận. Vì thế ta sử dụng cấu trúc device này để quyết định mã hay giải mã gói tin gửi đi hay nhận được.

Người lập trình áp dụng cơ chế con trỏ hàm trong ngôn ngữ C để thi hành nhân Linux một cách linh hoạt. Bằng chứng là Linux hỗ trợ nhiều kiểu hệ thống file, nhiều họ giao thức, và mang tính tương thích cao với các hệ thống khác.

Chính con trỏ hàm cho phép người lập trình thi hành cơ chế module đối với nhân Linux. Cụ thể, khi chèn module vào nhân thì con trỏ hàm trỏ tới module của họ, kể từ đó nhân Linux hoạt động với tích năng mới có trong module. Khi loại bỏ module, người lập trình cần thiết lập cho nhân hoạt động như lúc chưa nạp module.

Người viết module có thể lợi dụng luôn con trỏ hàm có sẵn trong nhân, cũng có lúc người lập trình phải tự tạo ra con trỏ hàm cho bài toán cụ thể của mình. Trong trường hợp thứ nhất người lập trình không phải biên dịch lại nhân, trong trường hợp thứ hai người lập trình phải tiến hành biên dịch lại nhân một lần duy nhất sau khi đã thêm con trỏ hàm vào nhân. Sau đó người lập trình chỉ còn phải tập trung phát triển hoàn thiện module.

Việc can thiệp mật mã vào tầng data link để bảo mật luồng dữ liệu truyền thông mạng được viết thành dạng module. Để có thể can thiệp vào nhân ở tầng mạng này, chúng ta phải tự tạo ra biến con trỏ hàm cho chính mình. Công việc được chia thành hai phần:

- Chèn các con trỏ hàm tại các vị trí thích hợp trong nhân và sau đó biên dịch lại nhân.

+ Thêm vào gần cuối file net/netsyms.c các dòng sau:

```
extern int (*encrypt_ptr)(struct sk_buff *);
EXPORT_SYMBOL_NOVERS(encrypt_ptr);
extern int (*decrypt_ptr)(struct sk_buff *);
EXPORT_SYMBOL_NOVERS(decrypt_ptr);
```

+ Chèn vào file net/core/dev.c (ngay trước hàm dev_queue_xmit()) dòng sau:

```
int (*encrypt_ptr)(struct sk_buff *) = 0;
```

+ Chèn vào file net/core/dev.c (ngay trước hàm net_bh( )) dòng sau:

```
int (*decrypt_ptr)(struct sk_buff *) = 0;
```

+ Chèn vào hàm dev_queue_xmit (trong file net/core/dev.c) đoạn mã in nghiêng sau:

```
int dev_queue_xmit(struct sk_buff *skb)
{
        struct device * dev = skb->dev;
        struct Qdisc *q;
        if (encrypt_ptr)
                (*encrypt_ptr)(skb);
        . . .
}
```

+ Chèn vào hàm net_bh (trong file net/core/dev.c) đoạn mã in nghiêng sau:

```
void net_bh(void)
{
        . . .
        handle_bridge(skb, type);
#endif
        if (decrypt_ptr)
                (*decrypt_ptr)(skb);
        . . .
}
```

+ Biên dịch lại nhân bằng các lệnh make menuconfig, make dep, make bzImage, copy nhân vào thư mục /boot , sửa file /etc/lilo.conf để chạy nhân vừa dịch xong.

- Viết mã cho module

```
#include <linux/module.h>
#include <linux/skbuff.h>
#include <linux/if_ether.h>
#include <linux/netdevice.h>
#include "blockcipher.h"
extern int (*encrypt_ptr)(struct sk_buff *);
extern int (*decrypt_ptr)(struct sk_buff *);

/* Hàm thực hiện kiểm tra gói tin và mã hoá trong hàm dev_queue_xmit() */

int encrypt_func(struct sk_buff *skb)
{
/* Nếu không phải gói tin chuyển qua eth0 thì không mã hoá */
        if (strcmp(skb->dev, "eth0") !=0)
                goto NOT_ENCRYPT;
/* Nếu gói tin chuyển qua eth0 thì mã hoá gói tin */
        Encrypt(skb); /* Gọi hàm mã hoá gói tin */
        return 1;
NOT_ENCRYPT:
        return 0;
}

/* Hàm thực hiện kiểm tra gói tin và giải mã trong hàm net_bh() */

int decrypt_func(struct sk_buf *skb)
{
/* Nếu gói tin nhận được không phải nhận từ eth0 thì không giải mã */
        if (strcmp(skb->dev, "eth0")
                goto NOT_DECRYPT;
/* Nếu gói tin nhận được từ card eth0 thì giải mã gói tin */
        Decrypt(skb);
        return 1;
NOT_DECRYPT:
        return 0;
```

```
}

/* Hàm khởi tạo module */
int init_module()
{
        EXPORT_NO_SYMBOLS;
        encrypt_ptr = encrypt_func;
        decrypt_ptr = decrypt_func;
        printk("Begining encryption !\n");
        return 0;
}
/* Hàm được gọi khi loại module */
void cleanup_module()
{
        encrypt_ptr = 0; / * Không dùng hàm mã hoá */
        decrypt_ptr = 0; /* Không dùng hàm giải mã */
        printk("Stopping encryption network frame \n");
}
```

+ Ta tạo ra make file dùng để biên dịch thành module:

```
CC=gcc
all:
        $(CC) –c datalink.c –O6 –D__KERNEL__ -DMODULE –Wall
clean:
        rm –f datalink.o core
```

+ Sau khi dịch xong module ta có thể dùng hai câu lệnh sau để liên kết hoặc gỡ bỏ module ra khỏi nhân:

```
        insmod datalink.o
        rmmod datalink.o
```

Chương 2
Phần mềm DL-Cryptor

1. Mã nguồn DL-Cryptor

Các file trong phần mềm DL-Cryptor:

TT	File mã nguồn	Mô tả
1	**Thư mục /usr/src/linux/net**	Chứa mã nguồn xử lý mạng trong nhân
1.1	/usr/src/linux/net/netsym.c	Chứa các khai báo về các hàm được sử dụng trong phần mã nguồn mạng Linux.
1.2	/usr/src/linux/net/core/dev.c	File này chứa các hàm xử lý mạng ở tầng data link trong nhân linux.
2	**DL_Cryptor/DL-module**	Chứa mã nguồn của module datalink
2.1	DL-module/blockcipher.h	File header chứa các khai báo dùng cho hàm thuật toán mã khối MK1.
2.2	DL-module/datalink.c	File chứa phần mã nguồn được dịch thành module datalink.o mã hóa dữ liệu mạng ở tầng data link.
3	**DL_Cryptor/KeyExchange-1.0/pubkey**	Chứa mã nguồn chương trình trao đổi khóa tự động dùng cho DL-Cryptor
3.1	pubkey/Makefile	
3.2	pubkey/config.h	
3.3	pubkey/dhkey.h	Chứa khai báo về biến và tham số sử dụng cho lược đồ trao đổi khóa Diffie-Hellman.
3.4	pubkey/lock.c	Chứa chương trình thực hiện khóa file để quản lý trình sự hoạt động của trình pkdl.
3.5	pubkey/main.c	Đây là tệp chứa hàm main( ) của trình pkdl.
3.6	pubkey/mk1.h	Chứa các tham số cho thuật toán mã khối MK1 dùng trong trao đổi khóa tự động.
3.7	pubkey/negotiate.c	Chứa các hàm thực hiện trao đổi gói tin.
3.8	pubkey/p_sha1.c	Chứa chương trình thực hiện hàm băm sha1, xác thực gói tin.
3.9	pubkey/packet.c	Chứa các hàm xử lý gói tin như đọc khóa Diffie-Hellman, chữ ký số ...
3.10	pubkey/kex.h	
3.11	pubkey/proto.c	Thực hiện việc trao đổi gói tin

2. Quá trình biên dịch và cài đặt DL-Cryptor

2.1 Các yêu cầu

- Máy tính cài đặt bản Linux Red Hat 6.2 (phiên bản nhân 2.2.14-5.0).
- Sử dụng phiên làm việc với người dùng root (siêu người dùng).

- Phải đảm bảo đã cài đặt gói kernel-source-2.2.14-5.0.i386.rpm và gói kernel-headers-2.2.14-5.0.i386.rpm. Ngoài ra các gói phục vụ cho quá trình biên dịch cũng phải được cài đặt.
- Người biên dịch phải biết cách cài đặt và chạy thử nhân mới không có hỗ trợ DL-Cryptor trước khi thực hiện quá trình này.
- Thực hiện copy hai file: netsym.c vào thư mục /usr/src/linux/net/ và dev.c vào thư mục /usr/src/linux/net/core/.

2.2 Dịch nhân mới

Để thực hiện dịch nhân mới ta theo các bước sau:

- Chuyển vào thư mục /usr/src/linux, thực hiện các lệnh.

```
[root@pvkh]# cd /usr/src/linux
[root@pvkh /linux]# make menuconfig
```

Lệnh make menuconfig sẽ đưa ra một danh sách các menu để lựa chọn tham số cấu hình nhân. Với phần mềm DL-Cryptor thì ta chỉ cần chú ý đến tùy chọn Networking options. Hãy chắc chắn rằng tùy chọn này bạn đã chọn. Sau đó bạn ghi lại cấu hình vừa chọn, thực hiện các lệnh tiếp theo.

```
[root@pvkh /linux]# make dep
[root@pvkh /linux]# make bzImage
```

- Sau khi đã dịch xong nhân mới bạn thực hiện các lệnh sau để cài đặt nhân:

```
[root@pvkh /linux]# cp arch/i386/boot/bzImage /boot/kernel_dlcryptor
```

Thêm các dòng sau vào cuối file /etc/lilo.conf:

```
image = /boot/kernel_dlcryptor
label = dlcryptor
read-only
root = /dev/hda1
```

(Chú ý: root = /dev/hda1 không nhất thiết phải đúng như vậy, mà căn cứ vào thư mục gốc / bạn cài vào phân vùng nào. Ví dụ, bạn cài vào phân vùng /dev/hda5 thì bạn phải khai báo thay dòng trên bằng dòng root = /dev/hda5).

Chạy lệnh:

```
[root@pvkh /linux]# lilo
```

Đến đây, quá trình biên dịch và cài đặt đã hoàn thành. Chúng ta phải thực hiện cài đặt lại máy để có nhân mới hỗ trợ phần mềm DL-Cryptor.

2.3 Biên dịch module datalink

Để biên dịch module datalink ta thực hiện các lệnh sau:

Chuyển vào thư mục /usr/src/DL_Cryptor:

```
[root@pvkh] cd /usr/src/DL_Cryptor
```

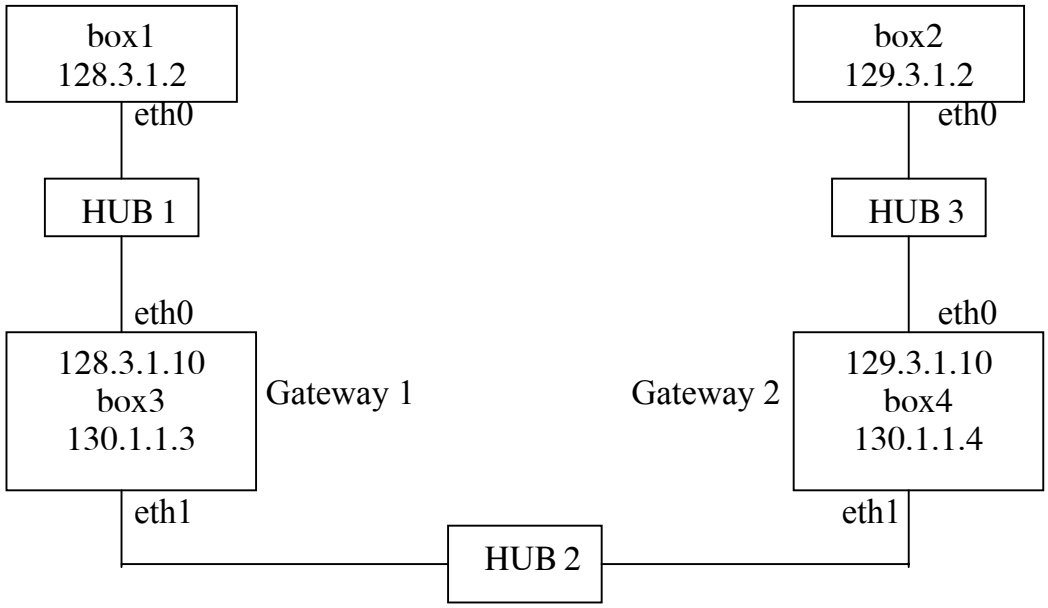
Ở thư mục DL_Cryptor ta thực hiện lệnh:

[root@pvkh /DL_Cryptor] make

Sau khi thực hiện câu lệnh này ta thấy trong thư mục này có module datalink.o. Copy file datalink.o vào thư mục /lib/modules/2.2.14-5.0/misc.

3. Thiết lập cấu hình cho DL-Cryptor

Chúng tôi lấy mô hình mạng tại phòng thí nghiệm an toàn mạng máy tính của PVKHMM làm ví dụ. Tùy thuộc vào mô hình mạng cụ thể mà khi cần bạn sẽ cấu hình hệ thống cho phù hợp.



Theo mô hình trên: box3 và box4 là hai gateway được nối với nhau qua ở hai giao diện mạng eth1 và được nối qua HUB 2. box1 và box2 là hai máy thuộc hai mạng con ở hai gateway box3 và box4. Chúng ta sẽ cài đặt phần mềm DL-Cryptor để bảo vệ luồng dữ liệu truyền thông mạng giữa hai gateway box3 và box4.

Sau khi đã thiết lập mạng theo mô hình ở trên, chúng ta cần thực hiện các bước sau:

- Chạy linuxconf hoặc ifconfig để thiết lập gateway mặc định cho box3 là box4 (130.1.1.4) và ngược lại.
- Đặt tùy chọn net.ipv4.ip_forward = 1 trong file /etc/sysctl.conf.
- Khởi động lại dịch vụ mạng: /etc/rc.d/init.d/network restart
- Thử dùng lệnh ping từ máy box1 sang máy box2, nếu ping được có nghĩa là bạn đã cấu hình hệ thống mạng thành công.

4. DL-Cryptor và trao đổi khóa tự động

Với phần mềm DL-Cryptor có thể có hai chế độ làm việc là manual (trao đổi khóa thủ công) và auto (trao đổi khóa tự động). Nhưng ở đây, chúng tôi chỉ giới thiệu phần DL-Cryptor với trao đổi khóa tự động. Ở mỗi phiên làm việc hai

máy chạy DL-Cryptor phải thực hiện trình trao đổi khóa kex để thỏa thuận khóa chung. Như vậy, mỗi phiên làm việc thì mỗi cặp gateway chạy DL-Cryptor lại có một khóa riêng khác biệt. Trình trao đổi khóa kex sẽ tự động chuyển khóa cho module datalink.o mỗi khi có phiên trao đổi khóa thành công giữa hai gateway.

- Tại box3:

Trong tệp /etc/hosts ta phải khai báo

box3 130.1.1.3
box 4 130.1.1.4

Trong tệp /etc/keyEx/pk/box4 chứa bộ tham số khóa công khai RSA của box4.

#PVKH: Public key for box 4 (RSA – public encryption).
peerModulo= 0x[nbox4]
peerPublicExponent=0x[ebox4]

Trong tệp /etc/keyEx/keyEx.priv chứa bộ tham số khóa công khai RSA của box3:

#PVKH: Public key for box3 (me this keep secret) (RSA – public encryption).
myModulo=0x[nbox3]
PrivateExponent=0x[dbox3]

Trong tệp /etc/inetd.conf ta phải thêm dòng khai báo:

pkdl stream tcp nowait root /usr/sbin/tcpd /usr/sbin/pkdl

Trong tệp /etc/services

pkdl 999/tcp #automatic key exchange for DL-Cryptor

- Tại box4:

Ở box4 chúng ta cũng có các file tương tự như ở box3.

Sau khi có các file khai báo như trên thì chúng ta chỉ việc thực hiện câu lệnh:

pkdl –c box4:999

5. Kiểm tra cài đặt và cấu hình

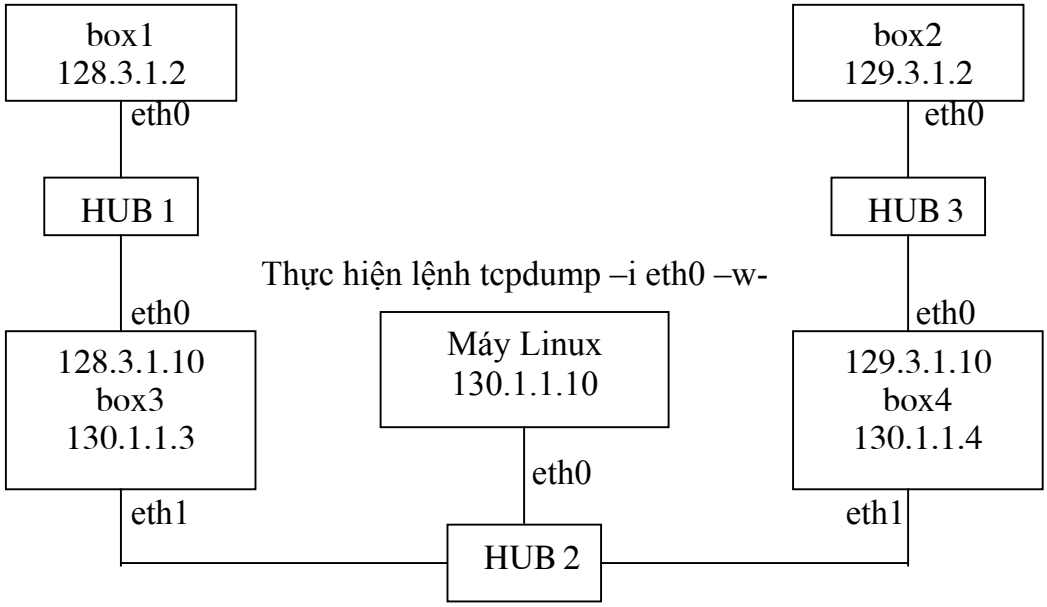
- Tại box3:

Sau khi cài đặt và cấu hình chúng ta thực hiện lệnh

pkdl –c box4:999

- Chúng ta thực hiện lệnh ping từ box1 sang box2. Nếu lệnh ping không lỗi thì qúa trình cấu hình và chạy phần mềm DL-Cryptor đã thành công.
- Nếu ping không được chúng ta cần phải kiểm tra các file cấu hình dùng cho chương trình pkdl.

- Dùng chương trình tcpdump ở Linux (hoặc windump trong Windows95/98) để kiểm tra việc mã hóa gói tin của DL-Cryptor. Tại box4 chúng ta thực hiện lệnh "ping 128.3.1.2 –p 414243", lệnh ping này sẽ gửi 3 tham số 41 (A), 42 (B), 43(C) sang cho máy box1. Nếu không mã hóa thì 3 ký tự này sẽ hiện trên màn hình của máy bắt gói. Nếu DL-Cryptor đã thực hiện mã hóa gói tin thì trên màn hình của máy bắt gói chúng ta không thấy 3 ký tự này nữa.



D. GIẢI PHÁP MẬT MÃ

Chương 1
Mã dữ liệu bằng mã khối

1. Mode hoạt động của mã khối MK1

Tương tự như thuật toán DES, thuật toán MK1 cũng có thể được dùng ở 4 mode là ECB (Electronic Codebook Mode), CFB (Cipher Feedback Mode), CBC (Cipher Block Chaining Mode), và OFB (Ouput Feedback Mode).

Với 2 mode là ECB và CBC thì độ dài dữ liệu cần mã hoá bởi thuật toán mã khối phải là bội của đầu vào hàm mã hoá (ví dụ như DES là 64 bít (8 byte), MK1 là 128 bít (16 byte)). Ở 2 mode là CFB và OFB thì độ dài dữ liệu mã hoá là tuỳ ý, không cần phải là chẵn khối. Như vậy, tuỳ vào từng trường hợp cụ thể mà ta áp dụng với từng mode cho phù hợp.

Việc sử dụng tầng datalink với chức năng hoạt động là chuyển gói tin từ mạng này sang mạng khác. Ta thấy datalink hoạt động ở tầng thứ 2 theo mô hình tham chiếu OSI. Ở tầng này datalink chỉ quan tâm tới địa chỉ MAC của gói tin. Ta thấy ở tầng này, việc can thiệp vào trình để thêm dữ liệu cho gói tin có độ dài chẵn khối đầu vào thuật toán mã hoá là một vấn đề hết sức tinh tế. Ở tầng này rất sát với tầng vật lý, vì vậy nếu người lập trình không biết can thiệp vào hàm cấp phát bộ nhớ cho cấu trúc struct sk_buff thì việc hệ thống đột ngột dừng do nhân không quản lý được bộ nhớ là điều thường xảy ra (kernel panic). Ở một khía cạnh khác, ở tầng thứ 2 này việc điều khiển tổng kiểm tra (checksum) cho gói tin ethernet là khó điều khiển. Hơn thế nữa, khi ta đã giải quyết được hai vấn đề trên, thì tốc độ mạng của máy tính linux chạy datalink sẽ trở nên chậm chạp rất nhiều. Vì vậy, để đưa hàm mã khối vào tầng datalink với 4 mode ở trên thì chúng ta thấy là việc sử dụng một trong 2 mode là CFB hoặc OFB là thuận tiện hơn cả. Trong bài viết này chúng tôi xin giới thiệu cụ thể hơn về mode OFB được ứng dụng với thuật toán mã khối MK1 được áp dụng trong trình DL-Cryptor.

- OFB mode (Output feedback mode - chế độ phản hồi đầu ra):

Trong chế độ OFB, dòng khoá sinh ra sẽ được xor với bản rõ (nó hoạt động tương tự như hệ mã dòng). OFB thực chất là một hệ mã dòng đồng bộ.

Khoá được sinh ra bằng cách lặp lại việc mã hoá khối 64-bit véc tơ khởi điểm ban đầu IV. Chúng ta định nghĩa z0=IV. Khoá $z_1z_2...z_i$ được sinh ra theo công thức sau:

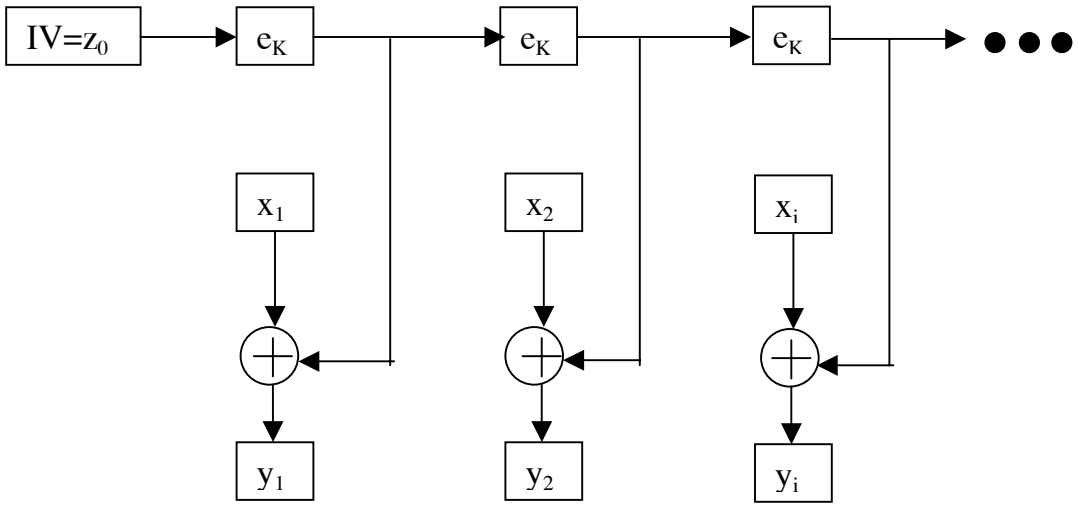
$$z_i = e_K(z_i - 1), i \geq 1$$

Chuỗi văn bản rõ $x_1x_2...x_i...$ được mã hoá bằng cách tính:

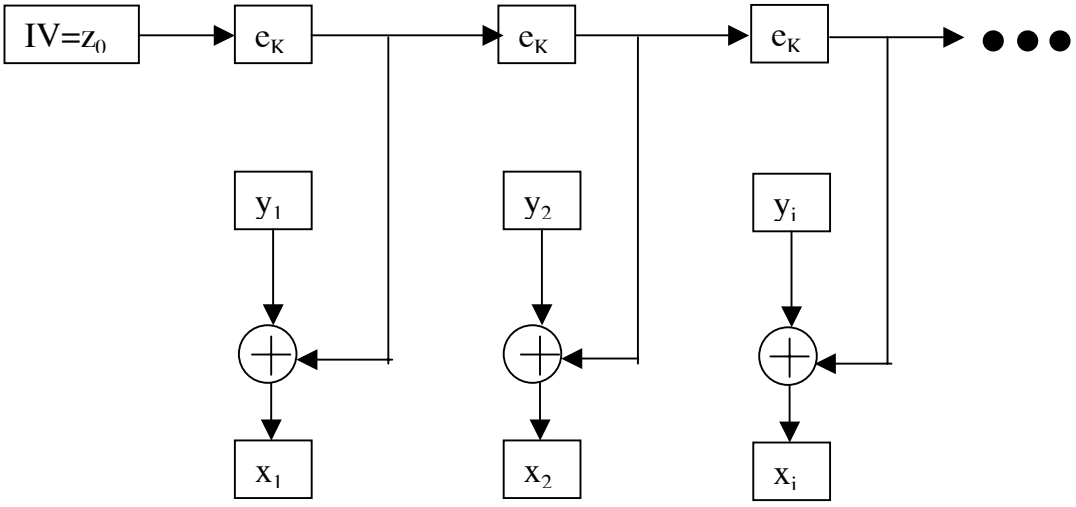
$$y_i = x_i \oplus z_i,\ i \geq 1$$

Sơ đồ thực hiện:

Mã hoá:



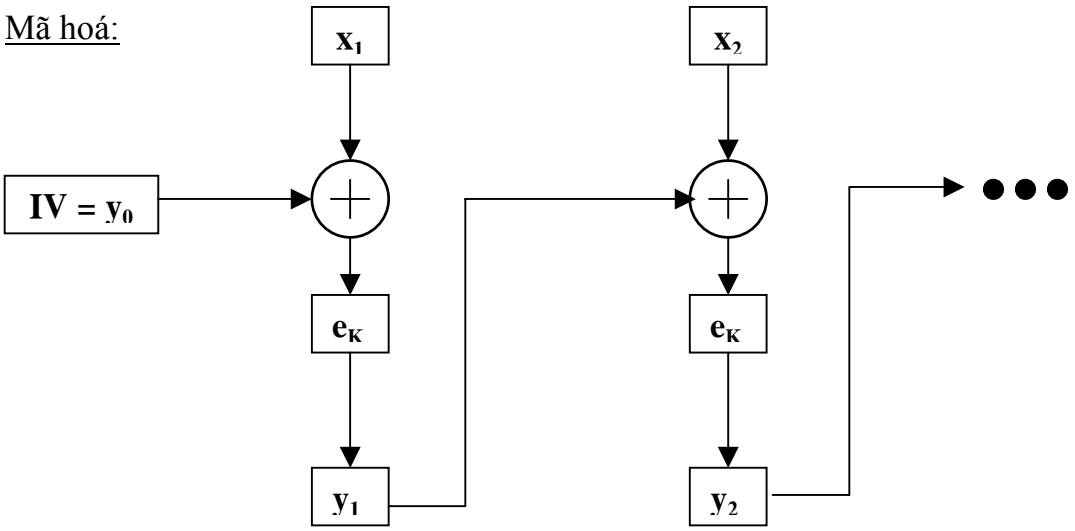
Giải mã:



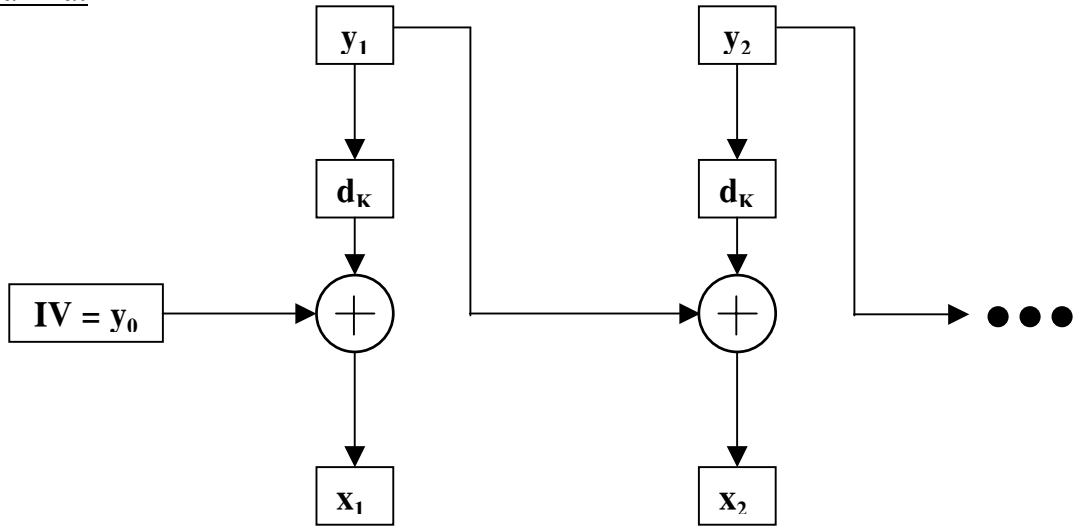
Trong phầm mềm IP-CRYPTO hàm mã khối MK1 được dùng ở mode CBC. Hàm mã khối MK1 với đầu vào sử dụng là 128 bít đầu ra là 128 bít, 512 bít khoá, và 128 bít véc tơ khởi điểm. Trong mode CBC, mỗi một khối bản mã y_i, được xor với khối bản rõ tiếp theo x_{i+1}, trước khi được mã hoá với khóa K. Cụ thể hợn, chúng ta bắt đầu với 128-bít véc tơ khởi điểm IV (Initialization Vector), và chúng ta định nghĩa y_0 = IV. Thì chúng ta tạo được bản mã $y_1y_2 \ldots y_i \ldots$ Theo công thức

$$y_i = e_K(y_{i-1} \oplus x_i),\ i \geq 1$$

Mã hoá:



Giải mã:



2.Khoá dùng trong MK1

Trong mode OFB, CBC thì để hai máy tính chạy datalink thực hiện mã hoá và giải mã thông suốt thì cả hai bên đều phải thoả thuận được khoá chung và véc tơ khởi điểm chung. Véc tơ khởi điểm trong hàm MK1 là 128 bít (16 byte), khoá là 512 bít (64 byte).

Việc thoả thuận khoá và véc tơ khởi điểm cho từng phiên làm việc sẽ được trình pkdl thực hiện theo lược đồ trao đổi khoá Diffie-Hellman dùng hệ mã khoá công khai RSA. Sau khi thực hiện trao đổi khoá xong thì hai bên máy linux datalink sẽ có véc tơ khởi điểm và khoá mã hoá dùng cho hàm mã hoá MK1 chung. Trong trình datalink của chúng tôi. Khi khởi động trình datalink thì hai máy linux datalink đã có một khoá chung thoả thuận trước, khoá này được gán cứng trong module. Như vậy, khi thực hiện trao đổi khoá giữa hai máy datalink thì mọi dữ liệu truyền thông giữa hai máy này đã được mã hoá. Khoá thoả thuận xong theo lược đồ Diffie-Hellman sẽ được trình trao đổi khoá tự động pkdl đưa vào nhân cho module datalink. Như vậy, cứ mỗi phiên làm việc của hai máy datalink thì có véc tơ khởi điểm riêng và khoá riêng.

Trong phần mềm IP-CRYPTO việc thoả thuận khoá do trình keyingd thực hiện, thực ra trình keyingd gọi đến trình kex để trao đổi khoá. Khoá sau khi thoả thuận xong được ghi vào file cấu thình (ipsec.conf), sau đó sẽ có trình kex sẽ gọi chương trình để đưa khoá vào trong nhân cho module ipsec.

Chương 2
Trao đổi khoá tự động

1. Thủ tục trao đổi khoá có xác thực

Mục đích của thủ tục xác thực là cung cấp cho các bên tham gia liên lạc một sự tin tưởng rằng họ biết định danh đúng của phía bên kia. Trong trao đổi khóa có xác thực, mục đích thêm mà 2 bên nhận được là chia sẻ một khóa chung mà chỉ có họ biết. Khóa bí mật đó có thể được dùng sau đó để đảm bảo bí mật, toàn vẹn dữ liệu, hay cả hai. Trong bài này, chúng ta bàn về độ an toàn của các thủ tục xác thực dựa trên mật mã khóa công khai, kèm theo việc trao đổi khóa. Chúng ta giới hạn mối quan tâm của mình tới việc xác thực 2 bên, thay cho các thủ tục nhiều bên hay một bên. Chúng ta giả thiết rằng các kỹ thuật mật mã nằm ở phía dưới là không bị tổn thương, và giới hạn mối quan tâm của mình chỉ ở phần thủ tục. Kẻ địch (người tấn công, người xâm nhập, đối phương) có thể xem tất cả các thông tin được trao đổi, có thể xóa, thay đổi, chèn hay có thể đổi hướng thông tin, có thể khởi đầu cuộc trao đổi với bất kỳ bên nào, và có thể sử dụng lại thông tin từ những cuộc liên lạc trước đó.

Việc thiết kế thủ tục mật mã nói chung, thủ tục xác thực nói riêng là rất hay mắc lỗi. Trong các tài liệu có rất nhiều thủ tục được tìm thấy có chứa lỗi về độ an toàn mức độ từ nhẹ tới rất nặng. Hơn nữa, bên cạnh độ an toàn, thực tế cho thấy rằng nhiều thủ tục đã được công bố chứa những cái dư thừa hoặc là không hiệu quả theo quan điểm số lần trao đổi thông tin cần làm, số các phép toán mật mã đòi hỏi (yêu cầu công suất tính toán cao), hoặc số lượng hay kiểu của các trường được yêu cầu trong các thông tin được trao đổi. Những điều này thúc đẩy việc tìm kiếm các thủ tục xác thực đơn giản, yêu cầu tối thiểu số lần trao đổi, số ít các trường trong mỗi thông điệp hay thẻ, số ít các phép toán mật mã. Những suy nghĩ đó đã thúc đẩy công trình này trên các thủ tục dựa vào khóa công khai.

Chúng ta quan tâm tới cả việc xác thực và cả việc trao đổi khóa. Nên chấp nhận xem xét 2 chủ đề này cùng với nhau chứ không tách biệt. Các thủ tục cung cấp việc xác thực nhưng không trao đổi khóa sẽ bị tổn thương bởi kẻ địch đợi cho đến khi việc xác thực đã hoàn thành rồi mới chiếm một đầu của kênh liên lạc. Tấn công này vẫn có tác dụng do việc trao đổi khóa độc lập với việc xác thực. Việc trao đổi khóa cần liên kết với việc xác thực sao cho các bên tin rằng khóa được trao đổi (có thể được dùng cho việc bảo mật hay xác thực và như vậy giữ cho *tính xác thực sống động*) thực chất là cái được chia sẻ với đối tác được xác thực, chứ không phải với một kẻ giả mạo. Với các lý do này, cần phải luôn nhớ tới việc trao đổi khóa trong khi thiết kế và phân tích các thủ tục xác thực.

Chúng tôi giới thiệu một thủ tục được gọi là trạm-tới-trạm (station-to-

station), kiểm tra nó chi tiết, bình luận các đặc tính của nó.Một số thủ tục có liên quan cũng được bàn đến, và các thủ tục được đề nghị là có liên quan đến chúng. Chúng tôi kết thúc cùng với việc tóm tắt những nguyên tắc mà chúng tôi cảm thấy là quan trọng trong việc thiết kế các thủ tục xác thực.

2. Định nghĩa của thủ tục an toàn

Một xuất hiện cụ thể của thủ tục xác thực được coi như là một *vận hành (run)*. Trước khi trình bày định nghĩa của thủ tục an toàn, trước hết chúng ta xem xét các tính chất của cái mà chúng ta coi là một vận hành thành công (successful run). Trong một vận hành thành công, hai đối tác liên lạc, Alice và Bob, trao đổi một số thông điệp, khi kết thúc thì họ có được niềm tin vào nhận dạng của nhau và hơn nữa, có thể là họ chia sẻ một khóa chung mà chỉ có họ biết. Với mỗi lần vận hành được thực hiện xong, mỗi bên hoặc là *chấp nhận* hoặc là *bác bỏ* nhận dạng của bên kia, và có thể làm thêm việc trao đổi khóa. Trong mỗi lần vận hành thành công, việc vận hành được thực hiện xong và cả hai bên đều chấp nhận.

Tính chất 1 của vận hành thành công: Cả Alice và Bob *chấp nhận* nhận dạng của phía bên kia. Nếu việc xác thực bao gồm cả trao đổi khóa thì cả hai cũng *chấp nhận* khóa đã được trao đổi.

Tính chất thứ hai mà vận hành thành công có liên quan là nhật ký của việc vận hành thủ tục (giả thiết các bên tham gia có ghi lại việc trao đổi). Để tiếp tục, chúng ta cần các định nghĩa để ý đến việc *trùng nhau (match)* khi áp dụng vào các nhật ký của việc vận hành.

Việc trùng các thông điệp: Chúng ta nói một thông điệp từ một nhật ký trùng với một thông điệp từ một nhật ký khác nếu một nhật ký ghi thông điệp như là tin đến, còn một nhật ký ghi thông điệp như là tin đi, và tất cả các trường có liên quan đến việc xác thực là như nhau trong cả hai nhật ký.

Việc định tính chất có liên quan đến xác thực cần phải cho phép các thông điệp có thể trùng nhau ngay cả khi chúng không trùng nhau từng bit. Qui cách ở đây là nếu thông điệp có chứa các trường không được ký, tức là về mặt mật mã không liên quan đến việc xác thực, thì sự khác biệt chỉ ở các trường này không làm ảnh hưởng việc làm cho thông điệp đáp ứng được định nghĩa của việc trùng.

Việc trùng nhật ký vận hành: Chúng ta nói rằng hai nhật ký của việc vận hành là trùng nhau nếu các thông điệp của chúng có thể phân chia thành tập của các thông điệp trùng (mỗi tập chứa 1 thông điệp từ mỗi nhật ký), các thông điệp xuất pháp từ một bên tham gia xuất hiện với cùng một thứ tự trong cả hai nhật ký, các thông điệp xuất phát từ bên tham gia còn lại cũng vậy. Để cho đơn giản, chúng ta không xem xét các thủ tục mà trong đó các thông điệp không đến đích theo thứ tự mà chúng đã được gửi đi.

Tính chất 2 của vận hành thành công: Nếu Alice và Bob cùng ghi lại quá trình trao đổi, thì nhật ký vận hành của họ trùng nhau.

Bây giờ đã đến lúc chúng ta định nghĩa xem cái gì là sự vận hành của một thủ tục xác thực đa phương (đối xứng hay phi đối xứng) không an toàn.

ĐỊNH NGHĨA 1: Một vận hành cụ thể của thủ tục được gọi là vận hành *không an toàn* nếu bất kỳ một bên tham gia nào trong vận hành, ví dụ như Alice, thực hiện thủ tục một cách trung thực, chấp nhận nhận dạng của phía bên kia, và một trong các điều kiện sau xảy ra:

- Tại thời điểm Alice chấp nhận nhận dạng của phía bên kia (trước khi cô ấy gửi hay nhận thông điệp tiếp theo), một phần hay toàn bộ nhật ký của phía bên kia không trùng với nhật ký của Alice.
- Khóa trao đổi được chấp nhận bởi Alice bị một một người khác biết, đây không phải là người mà Alice đã chấp nhận nhận dạng. (Điều kiện này không áp dụng cho việc xác thực không có trao đổi khóa).

Mục đích của kẻ thù địch là làm cho việc vận hành trở nên không an toàn. Mục đích của người thiết kế thủ tục là làm cho công việc của kẻ địch trở nên không thể (hay là không thể về mặt tính toán) trong mọi thể hiện của thủ tục (trong mọi lần vận hành). Ngược lại với Định nghĩa 1, chúng ta có định nghĩa của thủ tục xác thực đa phương an toàn (đối xứng hay phi đối xứng):

ĐỊNH NGHĨA 2: *Thủ tục an toàn* là thủ tục mà đối với chúng các điều kiện sau là đúng trong mọi trường hợp, khi mà một bên, ví dụ Alice, thực hiện thủ tục một cách trung thực và chấp nhận nhận dạng của phía bên kia:

- Tại thời điểm Alice chấp nhận nhận dạng của phía bên kia (trước khi cô ta gửi hay nhận thông điệp tiếp theo), một phần hay toàn bộ nhật ký vận hành của phía bên kia trùng với nhật ký vận hành của Alice.
- Đối với khóa trao đổi nếu được chấp nhận bởi Alice thì không thể về mặt tính toán một ai đó có thể tìm ra, ngoại trừ chính Alice và có thể là đối tác mà Alice đã chấp nhận nhận diện. (Điều kiện này không có đối với việc xác thực không có trao đổi khóa).

Trong khi các kỹ thuật phân tích hình thức có thể sử dụng thành công để khám phá các điểm yếu trong một số thủ tục xác thực, chứng minh tính đúng đắn là khó hơn nhiều, và phụ thuộc nhiều vào việc mô hình hóa đúng mục đích và các giả thuyết. Một kỹ thuật khác có thể dùng để khám phá điểm yếu là việc vét cạn đối với tấn công chèn (interleaving attack). Thật đáng tiếc, vì không có được chứng minh tuyệt đối về tính đúng đắn, niềm tin vào thủ tục chỉ có được khi các chuyên gia liên tục tiến hành phân tích nó và thất bại trong việc tìm ra điểm yếu.

3. Các đặc trưng mà thủ tục cần có

Bên cạnh tính an toàn, sau đây là các đặt tính khác cần có cho một thủ tục.

Độ an toàn chuyển tiếp hoàn thiện (Perfect Forward Secrecy-PFS). Thủ tục trao đổi khóa có xác thực cung cấp độ an toàn chuyển tiếp hoàn thiện nếu việc để lộ tài liệu khóa bí mật thời hạn dài (long-term secret key material) không làm tổn hại đến độ an toàn của những khóa được trao đổi trong những lần vận hành trước. Tính chất độ an toàn chuyển tiếp hoàn thiện không áp dụng cho việc xác thực không có trao đổi khóa.

Xác thực trực tiếp. Trong một số thủ tục trao đổi khóa, việc xác thực không được thực hiện cho đến khi cả hai bên chứng minh tri thức về khóa chung bằng cách sử dụng nó trong liên lạc sau đó. Thủ tục như vậy gọi là *không trực tiếp (indirect)*. Nếu việc xác thực được thiết lập ngay sau khi kết thúc việc vận hành thủ tục thì thủ tục được gọi là *trực tiếp*. Một thủ tục gián tiếp có thể được sửa đổi để trở thành trực tiếp bằng cách thêm việc trao đổi một thông điệp đã biết hay thông điệp có độ dư được mã bằng khóa đã trao đổi. Với xác thực không có trao đổi khóa, thủ tục gián tiếp không cung cấp tính bảo mật vì không có bên nào có thể chấp nhận nhận dạng của bên kia.

Không có tem thời gian. Trong khi tem thời gian là quen thuộc cho mục đích quản lý và tài liệu hóa, trong thực hành cần phải không dựa vào việc sử dụng nó để đảm bảo độ mật của thủ tục xác thực.

Để sử dụng tem thời gian cho xác thực, tất cả các bên tham gia đều phải duy trì đồng hồ của mình được đồng bộ thường xuyên một cách an toàn với nguồn thời gian tin cậy. Giữa những lần đồng bộ với nguồn thời gian tin cậy, thời gian ở từng trạm có thể khác nhau. Hai bên, Alice và Bob, cần cho phép một cửa sổ thời gian cho tem thời gian để bù lại những chênh lệch đồng hồ cục bộ và việc thông điệp đi trên mạng cũng tốn thời gian. Alice sẽ chấp nhận tem thời gian từ Bob nếu nó nằm trong cửa sổ xung quanh thời gian tại đồng hồ cục bộ của Alice và thêm nữa là Bob phải chưa sử dụng giá trị thời gian này trước đây. Alice có thể lưu trữ tất cả giá trị thời gian được sử dụng bởi tất cả các bên mà nằm trong cửa sổ hiện nay của cô ta (điều này có thể là không thực tế trong một số môi trường truyền thông) hoặc Alice có thể lưu trữ thời điểm cuối cùng được sử dụng bởi mỗi bên và kiên quyết đòi hỏi tăng giá trị thời gian từ mỗi bên tham gia. Tuy nhiên, trong trường hợp giá trị thời gian tăng thực sự, nếu Bob sử dụng thời điểm t ở tương lai xa vì một lý do nào đó (sự chênh lệch đồng hồ lớn hoặc đồng bộ sai với nguồn thời gian tin cậy), thì Bob không thể liên lạc với Alice cho đến khi thời điểm t nằm trong cửa sổ của Alice. Để tránh điều này, Alice cần phải lưu trữ thời điểm t và không cập nhật nhật ký của mình đối với giá trị thời gian cuối cùng được sử dụng bởi Bob. Điều này có thể dẫn đến việc phải chọn số lượng lớn các dữ liệu được lưu

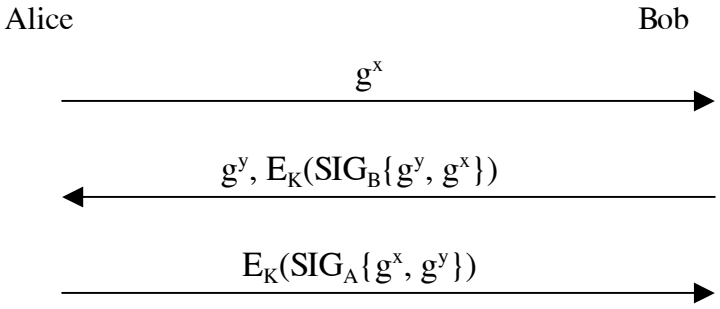
trữ, hy sinh khả năng truyền thông hoặc hy sinh độ an toàn. Liên quan đến khả năng truyền thông, nếu đồng hồ cục bộ của 2 bên rất không đồng bộ với nhau thì hai bên không thể liên lạc. Điều này làm cho những ai lo lắng tới khả năng truyền thông muốn cửa sổ thời gian rộng, như vậy làm tăng yêu cầu lưu trữ. Trong khi tem thời gian được coi là tiện lợi từ quan điểm lý thuyết, nó làm nảy sinh nhiều vấn đề thực tế. Những thủ tục dựa trên các thách thức ngẫu nhiên không gặp phải những khó khăn ấy.

4. Thủ tục STS (trạm-tới-trạm)

STS là một thủ tục trao đổi khoá theo lược đồ Diffie-Hellman, kèm theo việc trao đổi chữ ký xác thực. Trong phiên bản cơ sở của thủ tục, chúng ta giả thiết rằng các tham số được sử dụng để thiết lập khoá là cố định và mọi người đều biết.

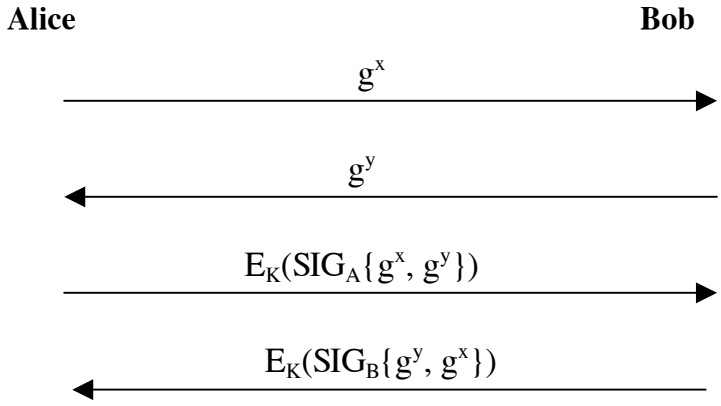
Thủ tục được bắt đầu bởi một bên, ví dụ, Alice, tạo ra một số ngẫu nhiên x, và gửi luỹ thừa g^x sang bên kia cho Bob. Bob tạo số ngẫu nhiên y, và sử dụng lũy thừa của Alice để tạo ra khoá trao đổi $K = g^{xy}$. Bob thực hiện việc ký lên các luỹ thừa bởi khoá K. Bob trả lời cùng với luỹ thừa g^y và một bản mã lên chữ ký của anh ta theo một thuật toán mã đối xứng thích hợp E. Alice nhận được sẽ tính khoá K, giải mã bởi K và kiểm tra chữ ký của Bob bằng cách dùng khoá công khai của Bob. Cuối cùng, tương tự, Bob kiểm tra chữ ký đã được mã hoá của Alice bằng cách dùng khoá K và khoá công khai của Alice.

Thủ tục STS



Có thể tạo ra một phiên bản đối xứng hơn cho thủ tục này, trong đó các bên trao đổi các luỹ thừa trước và sau đó trao đổi bản mã lên các chữ ký riêng biệt. Trong trường hợp này, cả Alice và Bob không cần biết ai bắt đầu cuộc gọi. Điều này là mong muốn, giống như các tình huống trong thực tế (đường điện thoại, đường truyền dữ liệu X.25) trong đó ở một mức độ cài đặt nào đấy, không rõ là bên nào bắt đầu trước. Điều này giải thích tại sao trong khuôn dạng chữ ký của mỗi bên thì luỹ thừa của chính người đó được đưa lên trước. Nếu luỹ thừa có cùng thứ tự trong cả hai chữ ký thì Alice và Bob cần tìm cách thoả thuận xem luỹ thừa của ai được đưa lên trước (ví dụ là dựa và qui định xem bên nào bắt đầu cuộc gọi).

Thủ tục STS đối xứng



5. Chương trình kex (key exchange) – Version 1.0

`kex` là chương trình trao đổi khoá có xác thực giữa hai bên tham gia liên lạc. Các bước trao đổi khoá tuân theo lược đồ Diffie-Hellman và sử dụng chữ ký số RSA.

`kex` thực hiện trao đổi khoá tự động giữa hai máy tính kết nối với nhau có sử dụng giao thức TCP/IP. Khi bắt đầu một quá trình trao đổi, hai chương trình `kex` sẽ được chạy trên hai máy, chúng kết nối với nhau thông qua giao thức TCP. `kex` ở mỗi bên sẽ gửi định danh và chữ ký số của mình cho phía bên kia. Mỗi bên sẽ kiểm tra chữ ký số nhận được bằng việc sử dụng khoá công khai RSA đã được phân phối trước của phía bên kia.

Để thực hiện một quá trình trao đổi khoá công khai RSA, trên mỗi máy phải có một cặp khoá (côngkhai/bímật). Khoá bí mật được lưu trong file /etc/keyEx/keyEx.priv. Thư mục /etc/keyEx/pk chứa khoá công khai của phía bên kia. Trên mỗi máy phải có một định danh (chúng tôi thường sử dụng tên máy) và nó phải được cho phía bên kia biết trước. Khoá công khai của phía bên kia được lưu trong thư mục /etc/keyEx/pk với tên file là tên định danh (tên máy) của phía bên kia. File keyEx.priv gồm hai tham số là myModulo và PrivateExponent, file khoá công khai cũng gồm hai tham số là peerModulo và peerPublicExponent.

Khi kết thúc quá trình trao đổi khoá, hai bên sẽ có một khoá ngẫu nhiên chung (với độ dài mặc định là 64 byte) để sử dụng cho các thuật toán mã hoá. Độ dài khoá tạo ra có thể thay đổi sao cho phù hợp với các thuật toán mã hoá. Ở đây chúng tôi tạo ra khoá có độ dài là 64 byte để sử dụng thuật toán Mã khối của Phân viện Khoa học Mật Mã.

Mô tả chi tiết quá trình trao đổi khoá: Chúng tôi sử dụng `kex` ở chế độ client-server, một bên luôn luôn đợi bên kia gửi đến. Đầu tiên, khi khởi động client sẽ tạo ra 56 byte dữ liệu ngẫu nhiên cùng với 8 byte nhãn thời gian (gettimeofday) gửi cho server và ngược lại. Hai bên sử dụng hàm DH_generate_key() của openssl để tạo ra khoá công khai Diffie-Hellman của mình. Cuối cùng thì hai bên sẽ nhận được khoá công khai của phía bên kia để tạo khoá Diffie-Hellman chung. Khi hai bên đã có một khoá Diffie-Hellman chung, chương trình sẽ dùng thuật toán SHA1 để tạo ra khóa giả ngẫu nhiên 64 byte. Kể từ đây, các gói tin trao đổi giữa hai máy (trao đổi xác thực và các tham số cần thiết khác) đã được mã hoá bằng thuật toán Mã khối (MK1). Hai bên sẽ trao đổi định danh cho nhau, chữ ký số được ký trên định danh của mỗi máy. Khoá tạo ra được lưu trong thư mục /etc/keyEx/run với tên file là định danh của phía bên kia.

Các bước trao đổi khoá

- Bước 1: Client gửi 64 byte dữ liệu NONCES và nhãn thời gian cho Server.
- Bước 2: Khi Server được kích hoạt nó cũng gửi cho Client 64 byte dữ liệu NONCES và nhãn thời gian.
- Bước 3: Khi Client nhận được dữ liệu, nó kiểm tra xem nếu là dữ liệu NONCES (packet type là PKT_NONCE) thì nó sẽ tạo ra khóa công khai Diffie-Hellman (DH) gửi cho Server.

```
case Snonce:
      if (pkttyp!=PKT_NONCE)
          goto stateerr;
      if (len<8)
          SendErrorRet("nonce too short");
        if (len>NONCESLEN+1)
            len=NONCESLEN+1;
        for (i=0; i<len-1; ++i)
            nonces[i]^=pkt[i+1];
        DH_generate_key(dhp);
      packetSendBN(fd, PKT_DHKEY, dhp->pub_key);
```

- Bước 4: Server tạo khoá công khai DH và gửi cho Client.
- Bước 5: Khi Client và Server nhận được khoá công khai DH của nhau, chúng sẽ tạo khoá DH chung, từ đó sử dụng thuật toán SHA1 để tạo khoá giả ngẫu nhiên 64 byte. Khoá này chính là khoá trao đổi được. Kể từ đây tất cả các gói tin trao đổi giữa hai bên đều được mã hoá với thuật toán MK1. Lúc này, Client sẽ gửi định danh của mình (hostname) cho Server.

```
case Sdhkey:
      if (pkttyp!=PKT_DHKEY)
          goto stateerr;
      if (mlock(buf, sizeof(buf))<0)
          Log(LOG_ERR, "handlePacket: mlock: %m");
      {
          BIGNUM *a=packetExtrBN(pkt, len);
          i=BN_cmp(a, dhp->pub_key);
```

```
          if (!i) {
           Log(LOG_ALERT, "handlePacket: Two equal DH
pubkeys???");
           abort(); /* something is VERY wrong */
          }
          if (DH_compute_key(buf, a, dhp)<0) {
           SSLprinterror(LOG_ERR);
           BN_free(a);
           SendErrorRet("internal bad DH");
          }
          BN_free(a);
      }
         /* Get all keys, switch to encrypted */
      i=getKeys(buf, DH_size(dhp), i);
      memset(buf, 0, sizeof(buf));
      if (i<0)
            SendErrorRet("handlePacket: getKeys failed");
      /* Send out identity next. */
      i=snprintf(buf, sizeof(buf), "%c%08x %s", PKT_IDENT,
              PKfingerprint(myKey), myIdentity);
      packetSend(fd, buf, i);
```

- Bước 6: Server nhận được định danh của Client gửi sang, nó sẽ gửi trả lại định danh của mình cho Client.
- Bước 7: Khi Client nhận được định danh của Server, nó sẽ tạo chữ ký số bằng cách ký lên định danh của Server rồi gửi cho Server.

```
      /* Sign our sent stuff. */
      buf[0]=PKT_SIGN;
      i=strlen(s)/2;
      pvkh_RSA_sign(buf+1, i);
      packetSend(fd, buf, strlen(buf));
```

- Bước 8: Server kiểm tra chữ ký số nhận được, nếu thành công, nó sẽ gửi chữ ký số của mình lên định danh của Client và gửi cho Client.

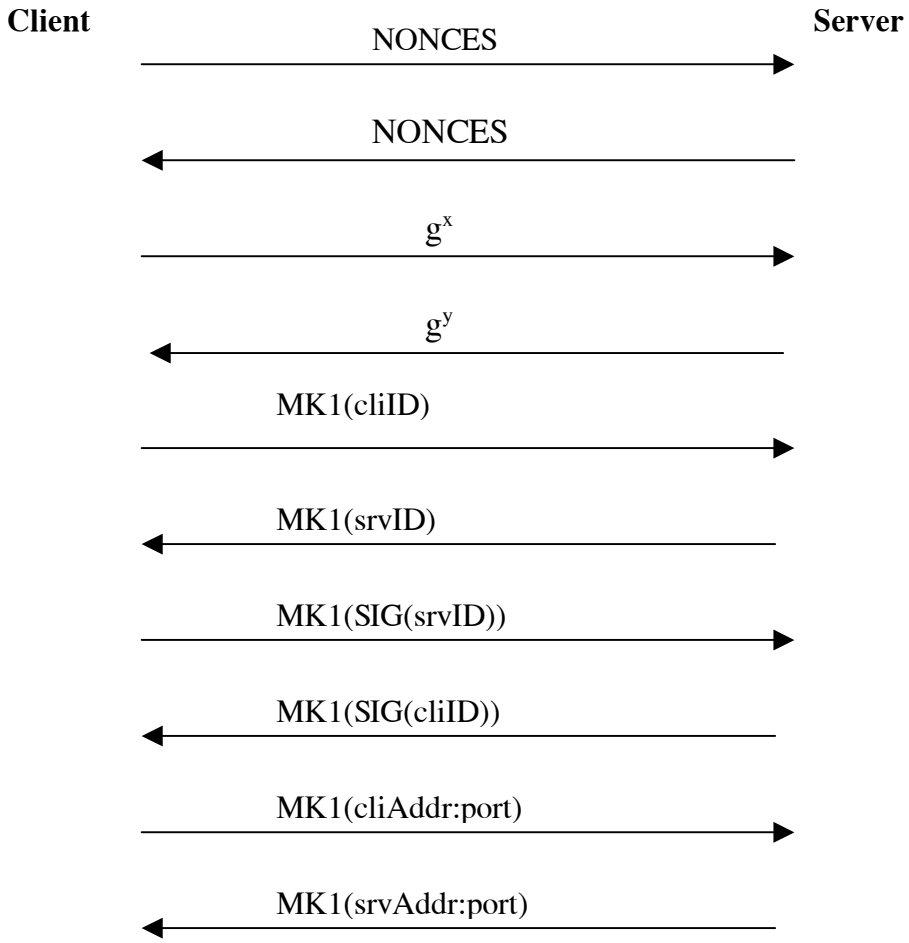
```
      if (pkttyp!=PKT_SIGN)
          goto stateerr;
      /* Verify the peer's signature. */
      if ((i=pvkh_RSA_vrfy(pkt+1, len-1, myKey))==1)
       {
          struct sockaddr_in sa;
          debug((DEB_PROTO, "Good signature"));
          if ((KEXsocket=getMySocket(&sa))<0)
          SendErrorRet("Could not obtain UDP socket");
          i=snprintf(buf, sizeof(buf), "%cme=%s:%d",
PKT_OPT_REQ,inet_ntoa(sa.sin_addr), ntohs(sa.sin_port));
          setOption("me", buf+4, OF_DEFAULT); /* later? */
          packetSend(fd, buf, i);
```

```
    return Sopt;
}
debug((DEB_PROTO, "Bad signature"));
SSLprinterror(LOG_ERR);
SendErrorRet("Signature check failed");
```

- Bước 9: Tương tự, Client kiểm tra chữ ký số của nhận được, nếu thành công, nó sẽ gửi gói tin chứa địa chỉ IP của mình cho Server. Việc các bên tham gia quá trình trao đổi khoá trao đổi thêm gói tin chứa địa chỉ IP của mình là rất quan trọng, làm tăng tính an toàn của thủ tục.
- Bước 10: Server gửi địa chỉ IP của mình cho Client. Khi Client nhận được địa chỉ IP của Server cũng là lúc hai bên thoả thuận xong khoá.

Nhận xét:

Việc hai bên trao đổi riêng biệt các gói tin chứa định danh, chữ ký số lên định danh của bên kia và gói tin chứa địa chỉ IP là không cần thiết. Chúng ta có thể gộp các tham số này lại và trao đổi một lần để giảm bớt số lần trao đổi. Công việc này chúng tôi sẽ thực hiện ở phiên bản sau.



Chữ ký số mà chúng tôi sử dụng được thực hiện đúng theo sơ đồ chữ ký số RSA. Giả sử `cli_sig` là chữ ký số được tạo ra bên Client, `cli_sig` sẽ được tính như sau:

$$cli\_sig = (srvID)^{srvPublicExponent} \bmod (srvModulo)$$

`cli_sig` sẽ được mã hoá với thuật toán MK1, rồi gửi cho Server. Server sẽ giải mã và có được `cli_sig`. Server kiểm tra chữ ký số như sau:

Tính: $chk\_sig = (srvID)^{srvPrivateExponet} \bmod (srvModulo)$. So sánh `chk_sig` xem có bằng với `cli_sig` hay không.

Qua sơ đồ trên chúng ta thấy rằng quá trình trao đổi cần rất nhiều bước, ta có thể thu gọn lại chỉ còn 3 hoặc 4 bước như thủ tục STS được trình bày ở trên. Tuy vậy, do thời gian có hạn nên chúng tôi chưa thực hiện được.

6. Sử dụng chương trình KEX

- Yêu cầu: Máy tính chạy hệ điều hành Linux RedHat 6.2 (nhân 2.2.14) hoặc RedHat 7.0 (nhân 2.2.16). Người sử dụng phải là root. Giả sử tên hai máy tính lần lượt là HOSTNAME1 và HOSTNAME2. Để biên dịch được chương trình nguồn, chúng ta cần bộ nguồn OpenSSL phiên bản 0.9.6 hoặc mới hơn.
- Tạo thư mục /etc/keyEx/pk, /etc/keyEx/run.
- Tạo hai bộ khoá công khai RSA cho hai bên như sau: (n1, e1, d1) cho HOSTNAME1 và (n2, e2, d2) cho HOSTNAME2.
- Trên máy HOSTNAME1 tạo hai file:
  - file thứ nhất: /etc/keyEx/keyEx.priv lưu n1 và d1 như sau:

```
# PVKH:  file /etc/keyEx/keyEx.priv
myModulo=0xn1
PrivateExponent=0xd1
```

  - file thứ hai: /etc/keyEx/pk/HOSTNAME2 lưu n2 và e2 như sau:

```
# PVKH:  file /etc/keyEx/pk/HOSTNAME2
peerModulo=0xn2
peerPublicExponent=0xe2
```

- Trên máy HOSTNAME2 tạo hai file tương tự như trên
  - file thứ nhất: /etc/keyEx/keyEx.priv lưu n2 và d2 như sau:

```
# PVKH:  file /etc/keyEx/keyEx.priv
myModulo=0xn2
PrivateExponent=0xd2
```

- file thứ hai: /etc/keyEx/pk/HOSTNAME1 lưu n1 và e1 như sau:

# PVKH: file /etc/keyEx/pk/HOSTNAME1
peerModulo=0xn1
peerPublicExponent=0xe1

Ví dụ: với cấu hình của chúng tôi, trên máy HOSTNAME1 là:

- file /etc/keyEx/pk/HOSTNAME2

peerModulo=0x89f5d9ec8170d69bfd403d9ed4037ff14a01abfdfbe94d5
ef64b73a32d22cfaf19f1a9b1b403b6d7b0f922c425085c9f34af3b64aaf
08af16a498d4814c69da3

peerPublicExponent=0x40c14214bfcaba17db7549cf934fe73f12345d9
855f8c9c10d5525f26a76545441537a74cc58867fdaf02d89296b29c6cff
f1b17fdd08eb4abc200b1b408c2130789f5d9ec8170d69bfd403d9ed4037
ff14a01abfdfbe94d5ef64b73a32d22cfaf19f1a9b1b403b6d7b0f922c42
5085c9f34af3b64aaf08af16a498d4814c69da3

Chú ý: khoá được ghi trên một dòng, trong file có thể có dấu cách, dấu '#'.

- file /etc/keyEx/keyEx.priv

myModulo=0x95441d0731fd9f5a1dc6dfb2f2eadec98024f8b013abdc772
d72346b9309cd59d3e68f92dff6e39051a792caac15a01fb5826f337524c
87e1bfedbb954ccef89

PrivateExponent=0x209c36eadf507badba36743d15073bd36503bd9c27
02f3f966befdd8ef2ac10c4cb15ddf7a5880db674bb3b0928ba5d17d54db
04436b234a6f37701e35330edd

- Trên máy HOSTNAME2:

  - file thứ nhất: /etc/keyEx/keyEx.priv lưu n2 và d2 như sau:

myModulo=0x89f5d9ec8170d69bfd403d9ed4037ff14a01abfdfbe94d5ef
64b73a32d22cfaf19f1a9b1b403b6d7b0f922c425085c9f34af3b64aaf08
af16a498d4814c69da3

PrivateExponent=0x48d1dd23dc94e7ab929cca519caeab6549aa4f1fd2
60baeacd9e1705851fa1609a25ec1b161e64a85af98d12b087b59534832f
eed40c7f53a815c0e7944ff0ff

  - file thứ hai: /etc/keyEx/pk/HOSTNAME1 lưu n1 và e1 như sau:

peerModulo=0x95441d0731fd9f5a1dc6dfb2f2eadec98024f8b013abdc7
72d72346b9309cd59d3e68f92dff6e39051a792caac15a01fb5826f33752
4c87e1bfedbb954ccef89

peerPublicExponent=0x40c43eb9f5d50f8c01bba8bf0ee8e6dae38b4de
9783734d139534bfbd4d5fac2072cea07ae0a8ff566d663561a45f74d73b
b51892458ef940137656e9e2949db4595441d0731fd9f5a1dc6dfb2f2ead

```
ec98024f8b013abdc772d72346b9309cd59d3e68f92dff6e39051a792caa
c15a01fb5826f337524c87e1bfedbb954ccef89
```

- copy file chạy `kex` vào thư mục /usr/sbin
- Lựa chọn cổng 10000 (phải đảm bảo cổng này chưa có dịch vụ nào sử dụng) cho kex bằng cách thêm dòng sau vào file /etc/services:
  kex 10000/tcp #port 10000 for kex
- Thêm dòng sau vào file /etc/inetd.conf
  kex stream tcp nowait root /usr/sbin/kex kex
  hoặc nếu sử dụng TCP Wrapper cho việc điều khiển truy nhập:
  kex stream tcp nowait root /usr/sbin/tcpd /usr/sbin/kex

Chú ý: *Với các phiên bản RedHat 7.0 hoặc 7.1, chương trình daemon inetd được thay thế bởi trình xinetd, file cấu hình cho dịch vụ này được đặt trong thư mục /etc/xinetd/. Khi này ta sẽ tạo thêm 1 file với tên là tên của dịch vụ cần chạy (ở đây là `kex`), nội dung như sau:*

```
service kex
{
     disable  = no
     user     = root        # người dùng được phép chạy
     port     = 10000  # cổng dịch vụ
     socket_type   = stream
     wait     = no
     server        = /usr/sbin/kex
}
```

Sau khi cấu hình xong với 2 file /etc/services và /etc/inetd.conf, ta nên khởi động lại máy để kích hoạt chương trình `kex`.

Lệnh chạy `kex`:

Trên HOSTNAME1:

kex -c HOSTNAME2:10000

Sau khi thực hiện lệnh này thì khoá tạo ra được lưu trong /etc/keyEx/run/HOSTNAME2

(Chý ý: Nếu chạy trên HOSTNAME2 thì thực hiện lệnh `kex -c HOSTNAME1:10000` - khoá tạo ra là file /etc/keyEx/run/HOSTNAME1).

Một vài tham số của KEX

`-c *hostname:port*`: kex chạy ở chế độ client-server, kết nối đến một máy có địa chỉ xác định.

`-t *timeout*`: Thiết lập thời gian chờ lần kết nối, mặc định là 60 giây.

`-k *KEYFILE*`: Chỉ ra file chứa private key và mymodulo, mặc định là /etc/keyEx/keyEx.prv

`-p *PROTO*`: Chỉ ra giao thức của KEX, hiện tại duy nhất là 1.

`-E`: Ghi các thông tin gỡ rối ra màn hình thay vì syslog (/var/log/message).

`-r *host*`: Xác định host là nơi mà các gói tin của KEX được routed qua. Tuỳ chọn này cần thiết khi kết nối TCP giữa hai máy thông qua một SOCKS hoặc một proxy khác.

Kiểm tra quá trình chạy `kex`

Sau khi chạy `kex` thành công, ở máy HOSTNAME1 khóa sẽ được lưu trong file /etc/keyEx/run/HOSTNAME2 (dòng key=), còn máy HOSTNAME2 khoá lưu trong /etc/keyEx/HOSTNAME1. Chúng ta hãy kiểm tra dòng `key=` trong hai file này, nếu giống nhau hoàn toàn thì quá trình trao đổi khoá thành công.

7. Các đặc tính của KEX

Các tham số thêm vào trong quá trình trao đổi khoá là rất cần thiết. Hiện tại tham số duy nhất được thêm vào là địa chỉ IP của mỗi máy. Việc sử dụng nhãn thời gian (timestamp) khi trao đổi khoá có tác dụng chống tấn công lặp lại. Tuy nhiên điều này đòi hỏi đồng hồ thời gian ở hai máy phải đồng bộ. Nhãn thời gian là một một số nguyên 32 bit (4 byte). Nó được lưu trong các gói tin trao đổi khoá. Bên nhận có thể sử dụng nhãn thời gian nhận được để loại bỏ gói tin nếu so sánh thời gian không khớp nhau. Người ta khuyên rằng nên để thời gian timeout không vượt quá 30 giây. Trong chương trình `kex`, nhãn thời gian chỉ được gửi lần đầu tiên, bên nhận cũng chưa sử lý việc kiểm tra. Nói tóm lại thủ tục này không sử dụng nhãn thời gian.

8. Trao đổi khoá tự động trong TransCrypt

KEX là một chương trình trao đổi khoá theo thuật toán Diffie-Hellman, kèm theo việc trao đổi chữ ký xác thực. Có thể coi thủ tục này an toàn như thủ tục STS bởi vì nó đảm bảo đầy đủ các điều kiện mà thủ tục STS nêu ra. Khoá tạo ra có thể được sử dụng cho các thuật toán mã hoá khác nhau. Các chương trình mã hóa sẽ đọc khoá từ file được tạo ra bởi `kex` để mã hoá dữ liệu.

Để thực hiện việc trao đổi khóa tự động trong TransCrypt, chúng tôi phải sửa đổi mã nguồn của chương trình KEX. Ý tưởng của chúng tôi là, sau khi chạy

chương trình `kex` để thoả thuận khoá, khoá sẽ được lưu vào một file. Chẳng hạn, trên máy HOSTNAME2 file này có dạng như sau:

arg=HOSTNAME1
peer=128.3.0.3
me=128.3.0.5
key=<64 byte>

Sau đó chương trình daemon của TransCrypt sẽ được gọi để đưa các tham số trong đó có khoá từ file này vào module của TransCrypt.

Việc gọi daemon được thực hiện trong hàm *ready()* của file *negotiate.c*:

```
{
#define TRANS_DAEMON "/usr/sbin/transcryptd-cb"
char ba[16];
char *na[]={TRANS_DAEMON, "-o", buf, "-s", ba, NULL};
if (KEXsocket>=0) {
    snprintf(ba, sizeof(ba), "%d", KEXsocket);
} else {
    /* should not happen */
    Log(LOG_ERR, "internal KEXsocket<0");
    na[3]=NULL;
}
execv(TRANS_DAEMON, na);
```

Giả sử chúng ta đang ở trên máy HOSTNAME1 thì dòng lệnh `execv(TRANS_DAEMON, na)` thực chất là gọi lệnh:

"/usr/sbin/transcryptd-cb -o /etc/keyEx/run/HOSTNAME2 - s KEXsocket"

Lệnh này hoàn toàn tương đương với việc chạy TransCrypt ở chế độ manual (khoá tĩnh).

Và như vậy, cứ mỗi lần`kex` được gọi thì nó sẽ tạo ra khoá mới và gọi daemon để nạp khoá mới cho module mã dịch TransCrypt

9. Trao đổi khoá tự động trong IP-Crypto

Trong phần mềm IP-Crypto thì việc trao đổi khoá tự động do chương trình keyingd đảm nhiệm. Trình keyingd gọi đến trình trao đổi khoá Kex để thực hiện việc thoả thuận khoá. Sau khi thoả thuận khoá xong thì trình Kex sẽ ghi khoá được thoả thuận vào file ipsec.conf, và sau đó gọi trình nạp khoá là manual để đưa khoá vào nhân.
Chương trình Kex ở đây được sửa đổi để có thể gọi trình manual mà không giết chết trình keyingd. Để làm được điều này chúng ta sinh ra một tiến trình con chạy cùng Kex và thực hiện gọi trình manual trong đó.

```
    pid_t pid;
    pid_t pid1;
    ...
    while (1) {
        if (n==0) {
        pid=fork(); /* Sinh tiến trình con */
        switch (pid) {
          case -1:
              printf("fork error");
              break;
          case 0:
              execv(RUN_FILE, nad);/* Thực thi tiến trình con */
          default:
              wait(0);
        }
    pid1=fork(); /* Sinh tiến trình con */
        switch (pid1) {
          case -1:
              printf("fork error");
              break;
          case 0:
              execv(RUN_FILE, nau); /* Thực thi tiến trình con*/
          default:
              wait(0);
        }

}
n=n+1;
break;
}
```

10. Trao đổi khoá tự động trong DL-Cryptor

KEX là một chương trình trao đổi khoá theo thuật toán Diffie-Hellman, kèm theo việc trao đổi chữ ký xác thực. Có thể coi thủ tục này an toàn như thủ tục STS bởi vì nó đảm bảo đầy đủ các điều kiện mà thủ tục STS nêu ra. Khoá tạo ra có thể được sử dụng cho các thuật toán mã hoá khác nhau. Các chương trình mã hóa sẽ đọc khoá từ file được tạo ra bởi `kex` để mã hoá dữ liệu hoặc có thể sử dụng ngay khoá vừa được trao đổi mà không cần phải ghi ra file (điều này đảm bảo an toàn hơn).

Để thực hiện việc trao đổi khóa tự động trong DL-Cryptor, chúng tôi phải sửa đổi mã nguồn của chương trình KEX. Ý tưởng của chúng tôi là, sau khi chạy chương trình `kex` để thoả thuận khoá. Lúc này, khoá sẽ không được lưu vào một file nữa, mà chúng tôi sẽ lưu vào một biến tĩnh (là key chẳng hạn).

Sau đó, khi chương trình `kex` thực hiện nó sẽ gọi lệnh *`insmod datalink.o key_encrypt=chuỗi_khoá_trong_biến_tĩnh_key`*.

Việc gọi lệnh *insmod* để nạp khoá được thực hiện trong hàm *ready()* của file *negotiate.c*:

```
      {
      #define RUN_FILE "sbin/insmod"
      char ba[16];
      char *na[]={RUN_FILE, "datalink.o", "key_encrypt=", key,
NULL};
      if (KEXsocket>=0) {
          snprintf(ba, sizeof(ba), "%d", KEXsocket);
      } else {
          /* should not happen */
          Log(LOG_ERR, "internal KEXsocket<0");
          na[3]=NULL;
      }
      execv(RUN_FILE, na);
```