Συσκευή κρυπτογράφησης για QEMU-KVM

Ομάδα 2
Άδωνις-Μάριος Τσεριώτης el17838
Γεώργιος-Σάββας Συρογιάννης el17140

Εργαστήριο Λειτουργικών Συστημάτων

Contents

1 Z1:Εργαλείο chat πάνω από TCP/IP sockets

1.1 Περιγραφή

Για το πρώτο ζήτημα της 3ης εργαστηριακής άσκησης μας ζητήθηκε να φτιάξουμε ένα εργαλείο chat πάνω απο TCP/IP sockets. Έχοντας διαχειριστεί ξανά τα sockets στην 1η εργαστηριακή άσκηση, δεν ήταν δύσκολη η κατανόηση του προβλήματος. Επιχειρήσαμε να φτιάξουμε ένα εργαλείο IRC το οποίο έχει έναν κεντρικό server στον ωκεανό και κάθε client μπορεί να συνδεθεί σε αυτόν εκτελώντας το πρόγραμμα ./irc_client [ip] [port]. Έπειτα, διαλέγει ένα username της επιλογής του και βλέπει ποιοί άλλοι χρήστες είναι online εκείνη την χρονική στιγμή. Διαλέγει έναν απο αυτούς και αρχίζει η μεταξύ τους επικοινωνία. Το IRC σκεφτήκαμε αρχικά να το κάνουμε με multi-threading. Δηλαδή, κάθε νέα σύνδεση να δημιουργεί ενα νέο νήμα το οποίο θα εκτελεί την επικοινωνία μεταξύ 2 πελατών. Αυτή η προσέγγιση όμως, μας φάνηκε δύσκολη και χρονοβόρα κι έτσι καταλήξαμε στην χρήση της **select()**. Συγκεκριμένα, ο *server* έχει ένα struct για τους users το οποίο μέσα περιέχει έναν buffer με το username του, το fd που του έχει δώσει ο server και ενα index το οποίο δείχνει με ποιόν user μιλάει. Αρχικά, αρχικοποιούμε το struct, κάνουμε establish την σύνδεση και ο server αρχίζει να ακούει για νέες συνδέσεις. Προσθέτουμε στο fd set που έχουμε ως όρισμα στην select τον server και όταν λάβει κάποιο request για σύνδεση, θα αρχικοποιήσουμε το struct user του χρήστη που προσπαθεί να συνδεθεί και θα του στείλουμε ένα μήνυμα να γράψει το username του. Θα συνεχίσει η επανάληψη και τώρα θα προστεθεί στο fd set και η νέα σύνδεση. Η select τώρα θα περιμένει να λάβει κάποιο request για I/O είτε απο τον χρήστη είτε απο τον server για κάποια νέα σύνδεση. Όταν ο χρήστης πληκτρολογήσει το username του, θα το λάβουμε και θα το συνέσουμε με το struct του συγκεκριμένου χρήστη. Άν λάβουμε κάποιο λάθος username input θα απολυθεί η σύνδεση. Στη συνέχεια, με σωστό username, θα προβάλλει ο server στην οθόνη του user τα username των υπόλοιπων χρηστών που είναι συνδεδεμένοι. Έτσι, δίνεται επιλογή στον χρήστη να επιλέξει έναν αριθμό απο το 1-N και να αρχίσει τη συνομιλία με τον χρήστη που προβάλλεται δίπλα απο τον εκάστοτε αριθμό. Ο client, απο την άλλη, κατά την εκτέλεση τρέχει την συνάρτηση connect και αν συνδεθεί στον server, ορίζει ένα νέο fd set το οποίο περιέχει μέσα το socket fd του server και το fd του standard input(0). Έπειτα, τρέχουμε τη select() και στον client έτσι ώστε να ελέγχουμε αν το input έρχεται απο τον χρήστη ή απο τον server και να μην είναι απαραίτητα διαδοχική η επικοινωνία. Αυτή είναι η κύρια υλοποίηση μας. Τα προβλήματα που αντιμετωπίσαμε ήταν σχετικά με τους buffers και τα null terminating strings. Οι buffers μας, γεμίζαν "σκουπίδια" και η επικοινωνία δεν γινόταν σωστά. Παρόλαυτα, διορθώθηκε το πρόβλημα και το IRC chat over TCP/IP ήταν πλήρως λειτουργικό.

1.2 Κώδικας

1.2.1 Header file

```
#ifndef _IRC_SERVER_H
#define _IRC_SERVER_H

struct
{
    char username[20];
    int sockfd;
    int speaks_with;
} user[10];

#define TCP_PORT    35001
#define BACKLOG     10
#define MAX_USERS   10

#define HELLO "Hello, please input username!\n"
#define USER_INERR "Please input a username with maximum 100 characters.\n"
#define ONLINE_USER "Please select user to speak to (Write 0-N):\n"
#define NO_ONLINE_USERS "No users available\n"


#endif
```

1.2.2 Server code

```
 /*
 * Simple IRC server that supports 10 clients
 *
 * George Syrogiannis <sirogiannisgiw@gmail.com>
 * Adonis Tseriotis <adonis.tseriotis@gmail.com>
 *
 */

#include <stdio.h>
#include <errno.h>
#include <ctype.h>
#include <string.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>
#include <netdb.h>

#include <sys/time.h>
#include <sys/types.h>
#include <sys/socket.h>

#include <arpa/inet.h>
#include <netinet/in.h>


#include "irc_server.h"

void clear_buff(char *buff,int n)
{
    int i;
    for(i=n; i<sizeof(buff); i++)
    {
        buff[i] = '\0';
    }
}
/* Insist until all of the data has been written */
ssize_t insist_write(int fd, const void *buf, size_t cnt)
{
        ssize_t ret;
        size_t orig_cnt = cnt;

        while (cnt > 0) {
                ret = write(fd, buf, cnt);
                if (ret < 0)
                        return ret;
                buf += ret;
                cnt -= ret;
        }

        return orig_cnt;
}

int getUserFromfd(int sockfd)
{
    int i;
    for(i=0; i<MAX_USERS; i++)
```

```
    {
        if(user[i].sockfd == sockfd)
        {
            return i;
        }
    }
    return -1;
}

int get_online_users(int connections, int me, int client_fd)
{
    char buff[100];
    int i;
    /* If no other users are available */
    if(connections == 1)
    {
        clear_buff(buff,0);
        strncpy(buff, NO_ONLINE_USERS, sizeof(buff));
        buff[sizeof(buff) - 1] = '\0';

        if((insist_write(client_fd, buff, sizeof(buff))) != sizeof(buff))
        {
            perror("write hello message");
            exit(1);
        }
        return -1;
    }

    for(i=-1; i<connections-1; i++)
    {
        /* Print initial message */
        if(i == -1)
        {
            clear_buff(buff,0);
            strncpy(buff, ONLINE_USER, sizeof(buff));
            buff[sizeof(buff) - 1] = '\0';
        }

        /* Print all other users */
        if(i != me && i != -1)
        {
            clear_buff(buff,0);
            strncpy(buff, user[i].username, sizeof(buff));
            buff[sizeof(user[i].username)-1] = '\n';
            buff[sizeof(buff) - 1] = '\0';
        }

        if(insist_write(client_fd, buff, sizeof(buff)) != sizeof(buff))
        {
            perror("write hello message");
            exit(1);
        }
    }
    return 0;
}

int handleUsernameInput(int clientfd, int serverfd, int connection, int all_connections)
```

```
{
    int n;
    char error[100];
    char username[20];

    clear_buff(username,0);
    n = read(clientfd, username, sizeof(username));
            /* Username input error handling */
    if (n < 0)
    {
        strncpy(error, USER_INERR, sizeof(error));
        error[sizeof(error) - 1] = '\0';
        if(insist_write(clientfd, error, sizeof(error)) != sizeof(error))
        {
            perror("write username usage message");
            exit(1);
        }
    }
    else if (n == 0)
    {
        return -1;
    }
    else
    {
        /* Save temp user to struct */
        username[n-2]='\0';
        strncpy(user[connection].username, username, n-1);
        //printf("\n\n\n\nSPEAKING WITH %s\n\n\n\n",user[connection].username);
        user[connection].speaks_with = serverfd;                                    //Set client ready to selec
        get_online_users(all_connections, connection,user[connection].sockfd);  //Display online user list
        return 1;
    }
    return 0;
}

/* Disconnect user and make speaking user go to home screen */
void disconnectUser(int connection, int serverfd, int all_connections)
{
    int n;
    char buff[100];
    int speaking_to,i;

    clear_buff(buff,0);
    n = sprintf(buff, "Sorry, %s has disconnected",user[connection].username);
    clear_buff(buff,n);
    /* Print disconnect message to other peer */
    if(insist_write(user[connection].speaks_with, buff, sizeof(buff)) != sizeof(buff))
    {
        perror("write disconnecting message");
        exit(1);
    }

    /* Find to who disconnecting user was speaking to */
    if((speaking_to = getUserFromfd(user[connection].speaks_with)) > 0)
    {
        user[speaking_to].speaks_with = serverfd;
        get_online_users(all_connections, connection, user[connection].speaks_with);
```

```
    }

    close(user[connection].sockfd);

    user[connection].sockfd = 0;
    /* Rectify user struct */
    for(i=connection; i<MAX_USERS-1; i++)
    {
        if(user[i].sockfd == 0 && user[i+1].sockfd != 0)
        {
            user[i].sockfd = user[i+1].sockfd;
            user[i+1].sockfd = 0;
        }
    }

}

/* Function to display that connection is made to users */
void display_connection(int me)
{
    char buff[80];
    int n;
    clear_buff(buff,0);
    n = sprintf(buff, "You are speakin to: %s\n",user[me].username);
    clear_buff(buff,n-1);

    if(insist_write(user[me].speaks_with, buff, n+1) != n+1)
    {
        perror("write disconnecting message");
        exit(1);
    }
}

/* Now chat */
int chat(int me)
{
    int n = 0;
    char buff[100];
    char wrbuff[124];
    /* Read message */
    clear_buff(buff,0);
    clear_buff(wrbuff,0);
    n = read(user[me].sockfd, buff, sizeof(buff));
    if(buff[0] == '\0')
    {
        return 0;
    }
    clear_buff(buff, n);
    if(n == 0)
    {
        return -1;
    }
    /* Print message where it's supposed to */
    printf("User %d with username: %s:",me, user[me].username);
    n = sprintf(wrbuff, "%s: %s", user[me].username, buff);
    clear_buff(wrbuff,n);
```

```
    if(insist_write(user[me].speaks_with, wrbuff, n+1) != n+1)
    {
        perror("write disconnecting message");
        exit(1);
    }

    clear_buff(buff,0);
    clear_buff(wrbuff,0);
    return n;
}



int main(void)
{
    char buff[100],userr[2];
    char addrstr[INET_ADDRSTRLEN];
    int serverfd, connection, maxfd, i,tempfd, event, selected_user;
    ssize_t n;
    socklen_t len;
    struct sockaddr_in sa;
    fd_set readfds;

    /* Initialise client fd array */
    for(i=0; i<MAX_USERS; i++)
    {
        user[i].sockfd = 0;
    }

    /* Ignore SIGPIPE signals */
    signal(SIGPIPE, SIG_IGN);

    /* Create TCP/IP socket */
    if ((serverfd = socket(PF_INET, SOCK_STREAM, 0)) < 0)
    {
        perror("socket");
        exit(1);
    }
    fprintf(stderr, "Created TCP socket\n");

    /* Bind to TCP well-known port */
    memset(&sa, 0, sizeof(sa));
    sa.sin_family = AF_INET;
    sa.sin_port = htons(TCP_PORT);
    sa.sin_addr.s_addr = htonl(INADDR_ANY);
    if (bind(serverfd, (struct sockaddr *) &sa, sizeof(sa)) < 0)
    {
        perror("bind");
        exit(1);
    }
    fprintf(stderr, "Bound TCP socket to port %d\n", TCP_PORT);

    /* Listen for incoming connections */
    if(listen(serverfd, BACKLOG) < 0)
    {
        perror("listen");
        exit(1);
```

```
}


/* Get hello message in buffer */
clear_buff(buff,0);
strncpy(buff, HELLO, sizeof(buff));
    buff[sizeof(buff) - 1] = '\0';

/* Initialize connection number */
connection = 0;
fprintf(stderr, "Waiting for an incoming connection...\n");

for(;;)
{
    /*  Remove all fds from set */
    FD_ZERO(&readfds);

    /* Add server socket to set */
    FD_SET(serverfd, &readfds);
    maxfd = serverfd;

    /* Add all active connections to set */
    for (i=0; i<MAX_USERS; i++)
    {
        tempfd = user[i].sockfd;

        /* If the client is online add it to the list */
        if(tempfd > 0)
            FD_SET(tempfd,&readfds);

        /* Update max fd */
        if(tempfd > maxfd)
        {
            maxfd = tempfd;
        }
    }

    /* Wait for some event to take place */
    event = select(maxfd+1, &readfds, NULL, NULL, NULL);

    if((event < 0) && (errno != EINTR))
    {
        printf("Something wrong with select\n");
    }


    /* Accept an incoming connection */
    if (FD_ISSET(serverfd, &readfds))
    {
        len = sizeof(struct sockaddr_in);
        if ((user[connection].sockfd = accept(serverfd, (struct sockaddr *)&sa, &len)) < 0)
        {
            perror("accept");
            exit(1);
        }
        /* Initialize user struct */
        user[connection].username[0] = '\0';
```

```
    user[connection].speaks_with = 0-serverfd;  //Choose username

    /* Print message for incoming connection */
    if(!inet_ntop(AF_INET, &sa.sin_addr, addrstr, sizeof(addrstr)))
    {
        perror("could not format IP address");
        exit(1);
    }
    fprintf(stderr, "Incoming connection from %s:%d\n",
        addrstr, ntohs(sa.sin_port));

    /* Request username */
    if(insist_write(user[connection].sockfd, buff, sizeof(buff)) != sizeof(buff))
    {
        perror("write hello message");
        exit(1);
    }

    /* Increment number of connections */
    connection++;
}

/* Find which client has requested IO */
for(i=0; i<connection; i++)
{
    tempfd = user[i].sockfd;

    if (FD_ISSET(tempfd, &readfds))
    {
        /* Input username */

        if (user[i].speaks_with == 0-serverfd)
        {
            if((handleUsernameInput(tempfd, serverfd, i, connection)) < 0)
            {
                disconnectUser(i, serverfd, connection);
            /* Fix to check all users if someone disconnects */
                i--;
                connection--; //Decrement number of connections
            }

            /* By now we have gotten username or user has disconnected */
        }

        /* Choose user to speak to */
        else if (user[i].speaks_with == serverfd)
        {
            n = read(user[i].sockfd, userr, sizeof(userr));
            if(n == 0)
            {
                disconnectUser(i,serverfd,connection);
                i--;
                connection--;
            }
            else
            {
                selected_user = atoi(userr);
```

```
                        /* Start connection */
                        user[i].speaks_with = user[selected_user].sockfd;
                        user[selected_user].speaks_with = user[i].sockfd;
                        /* Display connection */
                        display_connection(i);
                        display_connection(selected_user);
                    }
                }

                /* Chat with ur friends */
                else
                {
                    if(chat(i)<0)
                    {
                        disconnectUser(i,serverfd,connection);
                        i--;
                        connection--;
                    }
                }
            }
        }
    }
    return 0;
}
```

1.2.3 Client code

```
/*
 * Simple IRC client that speaks to implemented server
 *
 * George Syrogiannis <sirogiannisgiw@gmail.com>
 * Adonis Tseriotis <adonis.tseriotis@gmail.com>
 *
 */

#include <stdio.h>
#include <errno.h>
#include <ctype.h>
#include <string.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>
#include <netdb.h>

#include <sys/time.h>
#include <sys/types.h>
#include <sys/socket.h>

#include <arpa/inet.h>
#include <netinet/in.h>


#include "irc_server.h"

void clear_buff(char *buff,int n)
{
    int i;
    for(i=n; i<sizeof(buff); i++)
```

```
    {
        buff[i] = '\0';
    }
}

/* Insist until all of the data has been written */
ssize_t insist_write(int fd, const void *buf, size_t cnt)
{
        ssize_t ret;
        size_t orig_cnt = cnt;

        while (cnt > 0) {
                ret = write(fd, buf, cnt);
                if (ret < 0)
                        return ret;
                buf += ret;
                cnt -= ret;
        }

        return orig_cnt;
}

int main(int argc, char *argv[])
{
    int sockfd, port, event, maxfd;
    ssize_t n;
    char rbuff[1024],wrbuff[1024];
    char *hostname;
    struct hostent *hp;
    struct sockaddr_in sa;
    fd_set read_fds;

    if(argc != 3)
    {
        fprintf(stderr, "Usage: %s [hostname] [port]\n", argv[0]);
        exit(1);
    }

    hostname = argv[1];
    port = atoi(argv[2]);
    if(port > 65535 || port <= 0)
    {
        fprintf(stderr, "Usage: port must be between 1 and 65535");
        exit(1);
    }

    /* Create TCP/IP socket */
    if((sockfd = socket(PF_INET, SOCK_STREAM, 0)) < 0)
    {
        perror("socket");
        exit(1);
    }
    fprintf(stderr, "Created TCP socket\n");

    /* Look up remote hostname by DNS */
    if (!(hp = gethostbyname(hostname)))
    {
```

```
            printf("DNS lookup failed for host %s\n", hostname);
            exit(1);
    }

sa.sin_family = AF_INET;
sa.sin_port = htons(port);
memcpy(&sa.sin_addr.s_addr, hp->h_addr, sizeof(struct in_addr));
fprintf(stderr, "Connecting to remote host.... "); fflush(stderr);

if (connect(sockfd, (struct sockaddr *) &sa, sizeof(sa)) < 0)
{
    perror("connect");
    exit(1);
}
fprintf(stderr, "Connected.\n");

for(;;)
{
    FD_ZERO(&read_fds);

    FD_SET(STDIN_FILENO, &read_fds);
    FD_SET(sockfd, &read_fds);

    maxfd = (sockfd > STDIN_FILENO ? sockfd : STDIN_FILENO);

    event = select(maxfd+1, &read_fds, NULL, NULL, NULL);

    if((event < 0) && (errno != EINTR))
    {
        printf("Something wrong with select\n");
    }

    /* Accept message sent from server */
    if(FD_ISSET(sockfd, &read_fds))
    {
        n = read(sockfd, rbuff, sizeof(rbuff));
        if (n < 0) {
            perror("read");
            exit(1);
        }

        if (insist_write(0, rbuff, n) != n) {
            perror("write");
            exit(1);
        }
    }
    if (FD_ISSET(STDIN_FILENO, &read_fds))
    {
        n = read(1,wrbuff, sizeof(wrbuff));
        clear_buff(wrbuff,n);
        wrbuff[n] = '\0';

        if (insist_write(sockfd, wrbuff, n+1) != n+1)
        {
            perror("write");
            exit(1);
        }
```

```
        }
    }
    return 0;
}
```

2 Z2: Κρυπτογραφημένο chat πάνω από TCP/IP

2.1 Περιγραφή

Η λογική είναι ακριβώς η ίδια με το Z1 απλά εντάσσεται και το ζήτημα της κρυπτογράφησης. Η κρυπτογράφηση γίνεται με τη βοήθεια του cryptodev module. Με την εξοικείωση που είχαμε απο την προηγούμενη εργαστηριακή, εισάγαμε το module στον πυρήνα και με τη βοήθεια του οδηγού, κάναμε απαραίτητη χρήση του module. Πιο συγκεκριμένα, κάθε φορά που στέλνεται κάποιο μήνυμα απο τον server αρχίζουμε ένα νέο session στην κρυπτογραφική συσκευή και με κατάλληλη χρήση των pointers του module κάνουμε κρυπτογράφηση ή αποκρυπτογράφηση των buffers που περιέχουν τα μηνύματα. Ο κώδικας που έχουμε εισάγει σε σχέση με το Z1 είναι το άνοιγμα του αρχείου του οδηγού και στον server και στον client. Επίσης, στον client είναι 2 γραμμες encrypt όταν η select επιλέγει το stdin πριν σταλθεί το μήνυμα και decrypt όταν η select δέχεται input απο τον server. Στον server αντίστοιχα decrypt όταν λαμβάνεται ένα μήνυμα και encrypt πριν σταλθεί κάποιο μήνυμα. Οι συναρτήσεις decrypt και encrypt υλοποιούνται στο header file οπότε και αυτό παρατίθεται παρακάτω για να μην υπάρχει επανάληψη του κώδικα. Τα προβλήματα που αντιμετωπίσαμε ήταν πάλι σχετικά με τους buffers και τα "σκουπίδια" που εισάγονταν σε αυτούς.

2.2 Κώδικας

2.2.1 Header file

```
#ifndef _IRC_SERVER_H
#define _IRC_SERVER_H
#define TCP_PORT    35001
#define BACKLOG     10
#define MAX_USERS   10

#define KEY "kwstas1312geias"
#define IV "elanasouporebro"
#define KEY_SIZE        16
#define BLOCK_SIZE      16
#define DATA_SIZE       256
#define HELLO "Hello, please input username!\n"
#define USER_INERR "Please input a username with maximum 100 characters.\n"
#define ONLINE_USER "Please select user to speak to (0-N):\n"
#define NO_ONLINE_USERS "No users available"
#define TRY_IT_AGAIN "This user seems to be offline try again\n"

struct
{
    char username[20];
    int sockfd;
    int speaks_with;
    int active;
} user[10];

struct {
    unsigned char in[DATA_SIZE],
    encrypted[DATA_SIZE],
    decrypted[DATA_SIZE],
    iv[BLOCK_SIZE],
    key[KEY_SIZE];
}data;
```

```
#endif

/* Insist until all of the data has been read */
ssize_t insist_read(int fd, void *buf, size_t cnt)
{
        ssize_t ret;
        size_t orig_cnt = cnt;

        while (cnt > 0) {
                ret = read(fd, buf, cnt);
                if (ret < 0)
                        return ret;
                buf += ret;
                cnt -= ret;
        }

        return orig_cnt;
}

/* Insist until all of the data has been written */
ssize_t insist_write(int fd, const void *buf, size_t cnt)
{
        ssize_t ret;
        size_t orig_cnt = cnt;

        while (cnt > 0) {
                ret = write(fd, buf, cnt);
                if (ret < 0)
                        return ret;
                buf += ret;
                cnt -= ret;
        }

        return orig_cnt;
}

/* Common functions */
void clear_buff(char *buff,int n)
{
    int i;
    for(i=n; i<sizeof(buff); i++)
    {
        buff[i] = '\0';
    }
}

int encrypt (char *buff, int cfd)
{
    //int i = 0;
    struct session_op sess;
    struct crypt_op cryp;

    memset(&data, 0, sizeof(data));
    memset(&sess, 0, sizeof(sess));
        memset(&cryp, 0, sizeof(cryp));
    //memset(&data, 0, sizeof(data));
```

```
memcpy(data.key, KEY, sizeof(data.key));
memcpy(data.iv, IV, sizeof(data.iv));
strncpy((char *) data.in, buff, sizeof(data.in));


#ifdef CLIEN
    printf("Data input\n");
    for(i = 0; i < DATA_SIZE; i++)
    {
        printf("%c",data.in[i]);
    }
    printf("\n");
#endif

sess.cipher = CRYPTO_AES_CBC;
    sess.keylen = KEY_SIZE;
    sess.key = data.key;

if (ioctl(cfd, CIOCGSESSION, &sess)) {
            perror("ioctl(CIOCGSESSION)");
            return 1;
    }

cryp.ses = sess.ses;
    cryp.len = sizeof(data.in);
    cryp.src = data.in;
    cryp.dst = data.encrypted;
    cryp.iv = data.iv;
    cryp.op = COP_ENCRYPT;

#ifdef CLIEN
    printf("\nCryp.key:\n");
    for (i = 0; i < sizeof(IV); i++)
        printf("%c", cryp.iv[i]);
    printf("\n");
#endif

if (ioctl(cfd, CIOCCRYPT, &cryp)) {
            perror("ioctl(CIOCCRYPT) u bitch");
            return 1;
    }


#ifdef CLIEN
    printf("\nEncrypted data:\n");
    for (i = 0; i < DATA_SIZE; i++) {
        printf("%c", data.encrypted[i]);
    }
    printf("\n");
#endif

/* Finish sesh */
if (ioctl(cfd, CIOCFSESSION, &sess.ses)) {
            perror("ioctl(CIOCFSESSION)");
            return 1;
    }
```

```
        return 0;
}

int decrypt (char *buff, int cfd)
{
    //int i;
    struct session_op sess;
    struct crypt_op cryp;

    memset(&data, 0, sizeof(data));
    memset(&sess, 0, sizeof(sess));
        memset(&cryp, 0, sizeof(cryp));
    //memset(&data, 0, sizeof(data));

    memcpy(data.key, KEY, sizeof(data.key));
    memcpy(data.iv, IV, sizeof(data.iv));
    memcpy(data.encrypted, buff, sizeof(data.encrypted));

    sess.cipher = CRYPTO_AES_CBC;
        sess.keylen = KEY_SIZE;
        sess.key = data.key;

    /* Start session with driver */
    if (ioctl(cfd, CIOCGSESSION, &sess)) {
                perror("ioctl(CIOCGSESSION)");
                return 1;
        }

    cryp.ses = sess.ses;
        cryp.len = sizeof(data.encrypted);
    cryp.src = data.encrypted;
        cryp.dst = data.decrypted;
    cryp.iv = data.iv;
        cryp.op = COP_DECRYPT;

    /* Decrypt data */
        if (ioctl(cfd, CIOCCRYPT, &cryp)) {
                perror("ioctl(CIOCCRYPT)");
                return 1;
        }

    /* End session with driver
     * Decrypted data are in data.decrypted array
     */
    if (ioctl(cfd, CIOCFSESSION, &sess.ses)) {
                perror("ioctl(CIOCFSESSION)");
                return 1;
        }

    /* strcpy(buff, (char *) data.decrypted);
    buff[sizeof(data.decrypted)] = '\0';
    for(i=0; i<sizeof(data.decrypted); i++)
    {
        printf("This is the decrypted message:%c\n",buff[i]);
    } */
    #ifdef SERVER
```

```
        memcpy(buff,data.decrypted,sizeof(data.decrypted));
    #endif

    #ifdef CLIENT
        if (insist_write(STDOUT_FILENO, data.decrypted, sizeof(data.decrypted)) != sizeof(data.decrypted))
            {
                perror("write");
                exit(1);
            }
    #endif

        return 0;

}

int isNumber (char *c) {
    if (c[0] >= '0' && c[0] <= '9' && ((c[1] >= '0' && c[1] <= '9') || c[1] == '\n'))
        return 1;
    return 0;
}
```

3 Z3: Υλοποίηση συσκευής cryptodev με VirtIO

3.1 Περιγραφή

Σε αυτο το ζήτημα, χρειάστηκε να έρθουμε σε επαφή με την παραεικονικοποίηση και το πλαίσιο VirtIO. Αφού συνειδητοποιήσαμε τις παραπάνω έννοιες και φορτώσαμε το κατάλληλο patch στο qemu, χρειάστηκε να αλλάξουμε τον κώδικα του πυρήνα του qemu και συγκεκριμένα το αρχείο **virtio-cryptodev.c**. Πιο συγκεκριμένα, την συνάρτηση **vq_handle_output()**. Σε αυτήν την συνάρτηση, η οποία τρέχει στο backend δηλαδή στο userspace του hypervisor, περνάμε απο το frontend που τρέχει στο kernel space του guest τους buffers, τα struct και ότι άλλο προτείνεται στον οδηγο να γυρνάει κάθε system call το backend στο frontend. Δηλαδή, αυτή η συνάρτηση είναι υπεύθυνη να ανοίγει το αρχείο **/dev/crypto**, να παίρνει τα δεδομένα που του έχουμε περάσει μέσω των scatter gather lists που έχουμε εισάγει μέσα στο VirtQueue-δομή που μας διαθετει το VirtIO για επικοινωνιά guest hypervisor- με τη συνάρτηση virtque_pop() και να εκτελεί τα εκάστοτε syscalls στο module του hypervisor και να κλείνει το αρχείο **/dev/crypto**. Σε frontend επίπεδο, συμπληρώσαμε τον κωδικα που μας δόθηκε για την εισαγωγή του module, τη δημιουργιά των dedicated driver αρχείων μέσω των οποίων μπορούμε να έχουμε πρόσβαση στον driver στον guest ο οποίος επικοινωνεί με τον hypervisor. Με βάση την εμπειρία μας απο την προηγούμενη άσκηση, τα κενά του αρχείου **crypto-module.c** συμπληρώθηκαν εύκολα βάζοντας ένα spinlock για κλείδωμα των κρίσιμων σημείων. Αντιθέτως, το αρχείο **crypto-chrdev.c** μας ταλαιπώρησε γιατί στην αρχή δεν κάναμε τα πάντα allocate στο heap δημιουργώντας έτσι προβλήματα στο πέρασμα των pointers στο backend. Πιο συγκεκριμένα, στην συνάρτηση *crypto_chrdev_open()*, ανοίγουμε το αρχείο και κάνουμε allocate τα pointers που χρειάζεται να περάσουμε στο backend και τα βάζουμε σε scatter-gather lists με τη συνάρτηση **sg_init_one()**, προσθέτουμε αυτά στο virtqueue με τη συνάρτηση **virtqueue_add_sgs()** κρατώντας το spinlock και τα περνάμε στο backend μέσω του virtqueue με τη συναρτηση **virtqueue_kick()**. Έπειτα με buisy wait περιμένουμε να λάβουμε τα δεδομένα απο το backend αφού τα έχει επεξεργαστεί. Στην open() περιμένουμε να μας γυρίσει το fd του **/dev/crypto** που ανοίγει ο hypervisor στο backend αι το περναμε στο struct που έχουμε για το open file του cryptodev. Στην *crypto_chrdev_release()* αντίστοιχα, περνάμε το fd που έχει ανοιχτεί στο backend ακολουθώντας την ίδια διαδικασία και κάνουμε deallocate το struct που έχουμε για το open file. Με την ίδια ακριβώς λογική, υλοποιούμε τις ioctl κλήσεις. Το μόνο που έπρεπε να προσέξουμε αυτή τη φορά είναι τα ορίσματα που δεχόμαστε απο το userspace του guest στο οποίο τρέχουν τα προγράμματα client και server. Πρέπει να πάρουμε τα δεδομενα απο το διαφορετικό address space σ'αυτό που βρισκόμαστε, οπότε εκτελούμε το syscall **copy_from_user()** και αντίστοιχα όταν επιστρέφουμε τα δεδομενα το **copy_to_user()**. Κάτι άλλο που πρέπει να προσέξουμε όταν μεταφέρουμε structs στο backend πρέπει να περναμε σε διαφορετικό scatter-gather list κάθε attribute του struct και μετά να τα επανασυνδέουμε στο backend.

3.2 Κώδικας

3.2.1 Backend

```
/*
 * Virtio Cryptodev Device
 *
 * Implementation of virtio-cryptodev qemu backend device.
 *
 * Dimitris Siakavaras <jimsiak@cslab.ece.ntua.gr>
 * Stefanos Gerangelos <sgerag@cslab.ece.ntua.gr>
 * Konstantinos Papazafeiropoulos <kpapazaf@cslab.ece.ntua.gr>
 *
 */

#include "qemu/osdep.h"
#include "qemu/iov.h"
#include "hw/qdev.h"
#include "hw/virtio/virtio.h"
#include "standard-headers/linux/virtio_ids.h"
#include "hw/virtio/virtio-cryptodev.h"
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <crypto/cryptodev.h>

static uint64_t get_features(VirtIODevice *vdev, uint64_t features,
                             Error **errp)
{
    DEBUG_IN();
    return features;
}

static void get_config(VirtIODevice *vdev, uint8_t *config_data)
{
    DEBUG_IN();
}

static void set_config(VirtIODevice *vdev, const uint8_t *config_data)
{
    DEBUG_IN();
}

static void set_status(VirtIODevice *vdev, uint8_t status)
{
    DEBUG_IN();
}

static void vser_reset(VirtIODevice *vdev)
{
    DEBUG_IN();
}

static void vq_handle_output(VirtIODevice *vdev, VirtQueue *vq)
{
    VirtQueueElement *elem;
    unsigned int *syscall_type, *ioctl_cmd;
```

```
struct crypt_op *crypt;
struct session_op *sess;
long *host_return_val;
unsigned char *session_key,*src,*iv,*dst;
uint32_t *ses_id;

int *cfd, *openfd;

DEBUG_IN();

elem = virtqueue_pop(vq, sizeof(VirtQueueElement));
if (!elem) {
    DEBUG("No item to pop from VQ :(");
    return;
}

DEBUG("I have got an item from VQ :)");

syscall_type = elem->out_sg[0].iov_base;
switch (*syscall_type) {
case VIRTIO_CRYPTODEV_SYSCALL_TYPE_OPEN:
    DEBUG("VIRTIO_CRYPTODEV_SYSCALL_TYPE_OPEN");
    /* ?? */
    cfd = elem->in_sg[0].iov_base;
    *cfd = open("/dev/crypto",O_RDWR);

    if(*cfd < 0)
    {
        DEBUG("SOMETHING'S WRONG WITH FD");
    }
    break;

case VIRTIO_CRYPTODEV_SYSCALL_TYPE_CLOSE:
    DEBUG("VIRTIO_CRYPTODEV_SYSCALL_TYPE_CLOSE");
    /* ?? */
    openfd = elem->out_sg[1].iov_base;
    if(close(*openfd) < 0)
        DEBUG("ERROR ON CLOSE FUNC")
    break;

case VIRTIO_CRYPTODEV_SYSCALL_TYPE_IOCTL:
    DEBUG("VIRTIO_CRYPTODEV_SYSCALL_TYPE_IOCTL");
    /* ?? */
    cfd = elem->out_sg[1].iov_base;
    ioctl_cmd = elem->out_sg[2].iov_base;
    DEBUG("OK THROUGH HERE");
    DEBUG("OK HERE ALSO");
    switch (*ioctl_cmd)
    {
    case CIOCGSESSION:
        DEBUG("STARTING SESSION");
        host_return_val = elem->in_sg[1].iov_base;
        session_key = elem->out_sg[3].iov_base;
        sess = elem->in_sg[0].iov_base;

        sess->key = session_key;
        if ((*host_return_val = ioctl(*cfd, *ioctl_cmd, sess))) {
```

```
                DEBUG("FAILED IOCTL SSESS");
            }
            DEBUG("END OF SESSION START");
            break;

        case CIOCFSESSION:
            DEBUG("START OF FSESSION");
            host_return_val = elem->in_sg[0].iov_base;
            ses_id = elem->out_sg[3].iov_base;
            if((*host_return_val = ioctl(*cfd, *ioctl_cmd,ses_id)))
            {
                DEBUG("FAILED IOCTL FSESS");
            }
            DEBUG("END OF FSESSION");
            break;

        case CIOCCRYPT:
            DEBUG("IN CCRYPT");
            host_return_val = elem->in_sg[1].iov_base;
            crypt = elem->out_sg[3].iov_base;
            src = elem->out_sg[4].iov_base;
            iv = elem->out_sg[5].iov_base;
            dst = elem->in_sg[0].iov_base;

            crypt->src = src;
            crypt->dst = dst;
            crypt->iv = iv;

            *host_return_val = ioctl(*cfd, *ioctl_cmd,crypt);

            if(*host_return_val)
            {
                DEBUG("FAILED IOCT CRYPTIOGDsv");
            }
            DEBUG("LEAVING CCRYPT");
            break;

        default:
            DEBUG("You must have been mistaken mate.");
            break;
        }

        break;

    default:
        DEBUG("Unknown syscall_type");
        break;
    }

    virtqueue_push(vq, elem, 0);
    virtio_notify(vdev, vq);
    g_free(elem);
}

static void virtio_cryptodev_realize(DeviceState *dev, Error **errp)
{
    VirtIODevice *vdev = VIRTIO_DEVICE(dev);
```

```
    DEBUG_IN();

    virtio_init(vdev, "virtio-cryptodev", VIRTIO_ID_CRYPTODEV, 0);
    virtio_add_queue(vdev, 128, vq_handle_output);
}

static void virtio_cryptodev_unrealize(DeviceState *dev, Error **errp)
{
    DEBUG_IN();
}

static Property virtio_cryptodev_properties[] = {
    DEFINE_PROP_END_OF_LIST(),
};

static void virtio_cryptodev_class_init(ObjectClass *klass, void *data)
{
    DeviceClass *dc = DEVICE_CLASS(klass);
    VirtioDeviceClass *k = VIRTIO_DEVICE_CLASS(klass);

    DEBUG_IN();
    dc->props = virtio_cryptodev_properties;
    set_bit(DEVICE_CATEGORY_INPUT, dc->categories);

    k->realize = virtio_cryptodev_realize;
    k->unrealize = virtio_cryptodev_unrealize;
    k->get_features = get_features;
    k->get_config = get_config;
    k->set_config = set_config;
    k->set_status = set_status;
    k->reset = vser_reset;
}

static const TypeInfo virtio_cryptodev_info = {
    .name          = TYPE_VIRTIO_CRYPTODEV,
    .parent        = TYPE_VIRTIO_DEVICE,
    .instance_size = sizeof(VirtCryptodev),
    .class_init    = virtio_cryptodev_class_init,
};

static void virtio_cryptodev_register_types(void)
{
    type_register_static(&virtio_cryptodev_info);
}

type_init(virtio_cryptodev_register_types)
```

3.2.2 Frontend

```
/*
 * crypto-chrdev.c
 *
 * Implementation of character devices
 * for virtio-cryptodev device
 *
 * Vangelis Koukis <vkoukis@cslab.ece.ntua.gr>
 * Dimitris Siakavaras <jimsiak@cslab.ece.ntua.gr>
```

```
 * Stefanos Gerangelos <sgerag@cslab.ece.ntua.gr>
 *
 */
#include <linux/cdev.h>
#include <linux/poll.h>
#include <linux/sched.h>
#include <linux/module.h>
#include <linux/wait.h>
#include <linux/virtio.h>
#include <linux/virtio_config.h>

#include "crypto.h"
#include "crypto-chrdev.h"
#include "debug.h"

#include "cryptodev.h"

/*
 * Global data
 */
struct cdev crypto_chrdev_cdev;

/**
 * Given the minor number of the inode return the crypto device
 * that owns that number.
 **/
static struct crypto_device *get_crypto_dev_by_minor(unsigned int minor)
{
        struct crypto_device *crdev;
        unsigned long flags;

        debug("Entering");

        spin_lock_irqsave(&crdrvdata.lock, flags);
        list_for_each_entry(crdev, &crdrvdata.devs, list) {
                if (crdev->minor == minor)
                        goto out;
        }
        crdev = NULL;

out:
        spin_unlock_irqrestore(&crdrvdata.lock, flags);

        debug("Leaving");
        return crdev;
}

/*************************************
 * Implementation of file operations
 * for the Crypto character device
 *************************************/

static int crypto_chrdev_open(struct inode *inode, struct file *filp)
{
        int ret = 0;
        int err;
        unsigned int len, num_in, num_out;
```

```
struct crypto_open_file *crof;
struct crypto_device *crdev;
unsigned int *syscall_type;
int *host_fd;
struct scatterlist syscall_type_sg,host_fd_sg, *sgs[2];
unsigned long flags;

debug("Entering");

syscall_type = kzalloc(sizeof(*syscall_type), GFP_KERNEL);
*syscall_type = VIRTIO_CRYPTODEV_SYSCALL_OPEN;
host_fd = kzalloc(sizeof(*host_fd), GFP_KERNEL);
*host_fd = -1;

num_in = 0;
num_out = 0;

ret = -ENODEV;
if ((ret = nonseekable_open(inode, filp)) < 0)
        goto fail;

/* Associate this open file with the relevant crypto device. */
crdev = get_crypto_dev_by_minor(iminor(inode));
if (!crdev) {
        debug("Could not find crypto device with %u minor",
              iminor(inode));
        ret = -ENODEV;
        goto fail;
}

crof = kzalloc(sizeof(*crof), GFP_KERNEL);
if (!crof) {
        ret = -ENOMEM;
        goto fail;
}
crof->crdev = crdev;
crof->host_fd = -1;
filp->private_data = crof;

/**
 * We need two sg lists, one for syscall_type and one to get the
 * file descriptor from the host.
 **/
/* ?? */
debug("ALL GOOD UNTIL HERE");
sg_init_one(&syscall_type_sg, syscall_type, sizeof(*syscall_type));
sgs[num_out++] = &syscall_type_sg;

sg_init_one(&host_fd_sg, &crof->host_fd, sizeof(crof->host_fd));
sgs[num_out + num_in++] = &host_fd_sg;

debug("ENTERING SEMAPHORE");
/**
 * Wait for the host to process our data.
 **/
/* ?? */
spin_lock_irqsave(&crdev->lock,flags);
```

```
        err = virtqueue_add_sgs(crdev->vq, sgs, num_out, num_in,
                                &syscall_type_sg, GFP_ATOMIC);
        virtqueue_kick(crdev->vq);

        while (virtqueue_get_buf(crdev->vq, &len) == NULL)
                /* do nothing */;

        spin_unlock_irqrestore(&crdev->lock,flags);

        /* If host failed to open() return -ENODEV. */
        /* ?? */
        debug("OUT OF SEMAPHORE");
        if(crof->host_fd <0)
                return -ENODEV;

        debug("OMG IT OPENED fd = %d", crof->host_fd);
fail:
        debug("Leaving");
        return ret;
}

static int crypto_chrdev_release(struct inode *inode, struct file *filp)
{
        int err,ret;
        unsigned int len,num_out;
        unsigned int num_in;
        struct crypto_open_file *crof = filp->private_data;
        struct crypto_device *crdev = crof->crdev;
        struct virtqueue *vq = crdev->vq;
        struct scatterlist syscall_type_sg,host_fd_sg, *sgs[3];
        unsigned int *syscall_type;
        unsigned long flags;

        debug("Entering release");

        syscall_type = kzalloc(sizeof(*syscall_type), GFP_KERNEL);
        *syscall_type = VIRTIO_CRYPTODEV_SYSCALL_CLOSE;


        num_out = 0;
        num_in = 0;
        ret = 0;
        /**
         * Send data to the host.
         **/
        /* ?? */
        sg_init_one(&syscall_type_sg, syscall_type, sizeof(*syscall_type));
        sg_init_one(&host_fd_sg, &crof->host_fd, sizeof(crof->host_fd));

        sgs[num_out++] = &syscall_type_sg;
        sgs[num_out++] = &host_fd_sg;

        /**
         * Wait for the host to process our data.
         **/
        /* ?? */
```

```
        spin_lock_irqsave(&crdev->lock, flags);

        err = virtqueue_add_sgs(vq, sgs, num_out, num_in,
                                &syscall_type_sg, GFP_ATOMIC);
        virtqueue_kick(vq);

        while (virtqueue_get_buf(vq, &len) == NULL)
                /* do nothing */;

        spin_unlock_irqrestore(&crdev->lock, flags);

        kfree(crof);
        debug("Leaving");
        return ret;

}

static long crypto_chrdev_ioctl(struct file *filp, unsigned int cmd,
                                unsigned long arg)
{
        long ret = 0;
        int err;
        struct crypto_open_file *crof = filp->private_data;
        struct crypto_device *crdev = crof->crdev;
        struct virtqueue *vq = crdev->vq;
        struct scatterlist syscall_type_sg, /* output_msg_sg, input_msg_sg, */ host_fd_sg, ioctl_cmd_sg, se
                session_op_sg, host_return_val_sg, ses_id_sg, crypt_op_sg, src_sg, dst_sg, iv_sg,
                        *sgs[8];
        unsigned int num_out, num_in, len;
#define MSG_LEN 100
        unsigned int *syscall_type, *ioctl_cmd=NULL;
        long *host_return_val=NULL;
        struct session_op *sess=NULL;
        struct crypt_op *crypt=NULL;
        uint32_t *ses_id=NULL;
        unsigned char *src=NULL,*iv=NULL,*dst=NULL, *session_key=NULL;
        unsigned long flags;

        debug("Entering");

        /**
         * Allocate all data that will be sent to the host.
         **/
        /* output_msg = kzalloc(MSG_LEN, GFP_KERNEL);
        input_msg = kzalloc(MSG_LEN, GFP_KERNEL); */
        syscall_type = kzalloc(sizeof(*syscall_type), GFP_KERNEL);
        *syscall_type = VIRTIO_CRYPTODEV_SYSCALL_IOCTL;

        num_out = 0;
        num_in = 0;

        /**
         *  These are common to all ioctl commands.
         **/
        sg_init_one(&syscall_type_sg, syscall_type, sizeof(*syscall_type));
        sgs[num_out++] = &syscall_type_sg;
        /* ?? */
```

```
sg_init_one(&host_fd_sg, &crof->host_fd, sizeof(crof->host_fd));
sgs[num_out++] = &host_fd_sg;

/**
 *  Add all the cmd specific sg lists.
 **/
//sess = (struct session_op *) arg;
//crypt = (struct crypt_op *) arg;

switch (cmd) {
case CIOCGSESSION:
        debug("CIOCGSESSION");

        ioctl_cmd = kzalloc(sizeof(*ioctl_cmd), GFP_KERNEL);
        *ioctl_cmd = CIOCGSESSION;

        sess = kzalloc(sizeof(*sess), GFP_KERNEL);
        ret=copy_from_user(sess, (struct session_op *) arg, sizeof(*sess));
        if(ret)
        {
                debug("It's user session");
                return ret;
        }
        debug("Copied from user");

        session_key = kzalloc(sess->keylen, GFP_KERNEL);

        debug("Allocated key");

        ret = copy_from_user(session_key, sess->key, sess->keylen);
        if(ret)
        {
                debug("It's user session key");
                return ret;
        }
        debug("IF its here then i don't know");

        sg_init_one(&ioctl_cmd_sg, ioctl_cmd, sizeof(*ioctl_cmd));
        sgs[num_out++] = &ioctl_cmd_sg;

        sg_init_one(&session_key_sg, session_key, sess->keylen);
        sgs[num_out++] = &session_key_sg;

        sg_init_one(&session_op_sg, sess, sizeof(*sess));
        sgs[num_out + num_in++] = &session_op_sg;
        break;

case CIOCFSESSION:
        debug("CIOCFSESSION");

        ioctl_cmd = kzalloc(sizeof(*ioctl_cmd), GFP_KERNEL);
        *ioctl_cmd = CIOCFSESSION;

        ses_id = kmalloc(sizeof(uint32_t), GFP_KERNEL);

        ret = copy_from_user(ses_id, (uint32_t *) arg, sizeof(*ses_id));
        if(ret)
```

```
        {
                debug("It's user session id");
                return ret;
        }

        sg_init_one(&ioctl_cmd_sg, ioctl_cmd, sizeof(*ioctl_cmd));
        sgs[num_out++] = &ioctl_cmd_sg;

        sg_init_one(&ses_id_sg, ses_id, sizeof(*ses_id));
        sgs[num_out++] = &ses_id_sg;
        break;

case CIOCCRYPT:
        debug("CIOCCRYPT");

        ioctl_cmd = kzalloc(sizeof(*ioctl_cmd), GFP_KERNEL);
        *ioctl_cmd = CIOCCRYPT;

        crypt = kzalloc(sizeof(*crypt), GFP_KERNEL);
        debug("allocated crypt");

        if(copy_from_user(crypt, (struct crypt_op *) arg, sizeof(*crypt)))
        {
                debug("Its Black P from the crypt");
                ret = -10;
                goto out;
        }
        debug("got crypt from user");

        src = kzalloc(crypt->len,GFP_KERNEL);
        debug("allocated src");

        if(copy_from_user(src, crypt->src, crypt->len))
        {
                debug("Its src from the crypt");
                ret = -10;
                goto out;
        }
        debug("got src from user");

        iv = kzalloc(16,GFP_KERNEL);
        debug("allocated iv");

        if(copy_from_user(iv, crypt->iv, 16))
        {
                debug("Its iv from the crypt");
                ret = -10;
                goto out;
        }
        debug("got iv from user");

        dst = kzalloc(crypt->len,GFP_KERNEL);
        debug("allocated dst");

        sg_init_one(&ioctl_cmd_sg, ioctl_cmd, sizeof(*ioctl_cmd));
        sgs[num_out++] = &ioctl_cmd_sg;
```

```
        sg_init_one(&crypt_op_sg, &crypt, sizeof(*crypt));
        sgs[num_out++] = &crypt_op_sg;

        sg_init_one(&src_sg, src, crypt->len);
        sgs[num_out++] = &src_sg;

        sg_init_one(&iv_sg, iv, 16);
        sgs[num_out++] = &iv_sg;

        sg_init_one(&dst_sg, dst, crypt->len);
        sgs[num_out + num_in++] = &dst_sg;

        break;

default:
        debug("Unsupported ioctl command");

        break;
}

host_return_val = kmalloc(sizeof(long), GFP_KERNEL);
/* Also common for every cmd */
sg_init_one(&host_return_val_sg, host_return_val, sizeof(*host_return_val));
sgs[num_out + num_in++] = &host_return_val_sg;

/**
 * Wait for the host to process our data.
 **/
/* ?? */
/* ?? Lock ?? */
spin_lock_irqsave(&crdev->lock, flags);

err = virtqueue_add_sgs(vq, sgs, num_out, num_in,
                        &syscall_type_sg, GFP_ATOMIC);
virtqueue_kick(vq);

while (virtqueue_get_buf(vq, &len) == NULL)
        /* do nothing */;

spin_unlock_irqrestore(&crdev->lock, flags);

if(cmd == CIOCGSESSION)
{
                /* if(host_return_val)
                {
                        debug("ioctl(CIOCGSESSION)");
                        return 1;
                } */
                sess->key = ((struct session_op *) arg)->key;
                if(copy_to_user((struct session_op *) arg, sess, sizeof(*sess)))
                {
                        debug("its Cioccession return of the key");
                        ret = -14;
                        goto out;
                }
}
```

```
        else if(cmd == CIOCFSESSION)
        {
                        /* if(host_return_val)
                        {
                                debug("ioctl(CIOCFSESSION)");
                                return 1;
                        } */
                        debug("Finito");
        }

        else if(cmd == CIOCCRYPT)
        {
                        /* if(host_return_val)
                        {
                                debug("ioctl(CIOCCRYPT)");
                                return 1;
                        } */
                        if(copy_to_user(((struct crypt_op *) arg)->dst, dst, crypt->len))
                        {
                                debug("its Cioccrypt");
                                ret = -14;
                                goto out;
                        }
        }

out:
        kfree(dst);
        kfree(src);
        kfree(iv);
        kfree(crypt);
        kfree(sess);
        kfree(session_key);
        kfree(syscall_type);
        kfree(ioctl_cmd);
        kfree(ses_id);
        debug("Leaving");
        if(ret>0)
                ret=0;
        return ret;
}

static ssize_t crypto_chrdev_read(struct file *filp, char __user *usrbuf,
                                  size_t cnt, loff_t *f_pos)
{
        debug("Entering");
        debug("Leaving");
        return -EINVAL;
}

static struct file_operations crypto_chrdev_fops =
{
        .owner          = THIS_MODULE,
        .open           = crypto_chrdev_open,
        .release        = crypto_chrdev_release,
        .read           = crypto_chrdev_read,
        .unlocked_ioctl = crypto_chrdev_ioctl,
};
```

```
int crypto_chrdev_init(void)
{
        int ret;
        dev_t dev_no;
        unsigned int crypto_minor_cnt = CRYPTO_NR_DEVICES;

        debug("Initializing character device...");
        cdev_init(&crypto_chrdev_cdev, &crypto_chrdev_fops);
        crypto_chrdev_cdev.owner = THIS_MODULE;

        dev_no = MKDEV(CRYPTO_CHRDEV_MAJOR, 0);
        ret = register_chrdev_region(dev_no, crypto_minor_cnt, "crypto_devs");
        if (ret < 0) {
                debug("failed to register region, ret = %d", ret);
                goto out;
        }
        ret = cdev_add(&crypto_chrdev_cdev, dev_no, crypto_minor_cnt);
        if (ret < 0) {
                debug("failed to add character device");
                goto out_with_chrdev_region;
        }

        debug("Completed successfully");
        return 0;

out_with_chrdev_region:
        unregister_chrdev_region(dev_no, crypto_minor_cnt);
out:
        return ret;
}

void crypto_chrdev_destroy(void)
{
        dev_t dev_no;
        unsigned int crypto_minor_cnt = CRYPTO_NR_DEVICES;

        debug("entering");
        dev_no = MKDEV(CRYPTO_CHRDEV_MAJOR, 0);
        cdev_del(&crypto_chrdev_cdev);
        unregister_chrdev_region(dev_no, crypto_minor_cnt);
        debug("leaving");
}

#include <linux/sched.h>
#include <linux/types.h>
#include <linux/semaphore.h>
#include <linux/slab.h>
#include <linux/module.h>
#include <linux/spinlock.h>
#include <linux/virtio.h>
#include <linux/virtio_config.h>

#include "crypto.h"
#include "crypto-chrdev.h"
#include "debug.h"

struct crypto_driver_data crdrvdata;
```

```
static void vq_has_data(struct virtqueue *vq)
{
        debug("Entering");
        debug("Leaving");
}

static struct virtqueue *find_vq(struct virtio_device *vdev)
{
        int err;
        struct virtqueue *vq;

        debug("Entering");

        vq = virtio_find_single_vq(vdev, vq_has_data, "crypto-vq");
        if (IS_ERR(vq)) {
                debug("Could not find vq");
                vq = NULL;
        }

        debug("Leaving");

        return vq;
}

/**
 * This function is called each time the kernel finds a virtio device
 * that we are associated with.
 */
static int virtcons_probe(struct virtio_device *vdev)
{
        int ret = 0;
        struct crypto_device *crdev;

        debug("Entering");

        crdev = kzalloc(sizeof(*crdev), GFP_KERNEL);
        if (!crdev) {
                ret = -ENOMEM;
                goto out;
        }

        crdev->vdev = vdev;
        vdev->priv = crdev;

        crdev->vq = find_vq(vdev);
        if (!(crdev->vq)) {
                ret = -ENXIO;
                goto out;
        }

        /* Other initializations. */
        /* ?? */
        //SPINLOCK?
        spin_lock_init(&crdev->lock);

        /**
```

```
         * Grab the next minor number and put the device in the driver's list.
         **/
        spin_lock_irq(&crdrvdata.lock);
        crdev->minor = crdrvdata.next_minor++;
        list_add_tail(&crdev->list, &crdrvdata.devs);
        spin_unlock_irq(&crdrvdata.lock);
        debug("Got minor = %u", crdev->minor);

        debug("Leaving");

out:
        return ret;
}

static void virtcons_remove(struct virtio_device *vdev)
{
        struct crypto_device *crdev = vdev->priv;

        debug("Entering");

        /* Delete virtio device list entry. */
        spin_lock_irq(&crdrvdata.lock);
        list_del(&crdev->list);
        spin_unlock_irq(&crdrvdata.lock);

        /* NEVER forget to reset virtio device and delete device virtqueues. */
        vdev->config->reset(vdev);
        vdev->config->del_vqs(vdev);

        kfree(crdev);

        debug("Leaving");
}

static struct virtio_device_id id_table[] = {
        {VIRTIO_ID_CRYPTODEV, VIRTIO_DEV_ANY_ID },
        { 0 },
};

static unsigned int features[] = {
        0
};

static struct virtio_driver virtio_crypto = {
        .feature_table = features,
        .feature_table_size = ARRAY_SIZE(features),
        .driver.name =        KBUILD_MODNAME,
        .driver.owner =        THIS_MODULE,
        .id_table =        id_table,
        .probe =        virtcons_probe,
        .remove =        virtcons_remove,
};

/**
 * The function that is called when our module is being inserted in
 * the running kernel.
 **/
```

```
static int __init init(void)
{
        int ret = 0;
        debug("Entering");

        /* Register the character devices that we will use. */
        ret = crypto_chrdev_init();
        if (ret < 0) {
                printk(KERN_ALERT "Could not initialize character devices.\n");
                goto out;
        }

        INIT_LIST_HEAD(&crdrvdata.devs);
        spin_lock_init(&crdrvdata.lock);

        /* Register the virtio driver. */
        ret = register_virtio_driver(&virtio_crypto);
        if (ret < 0) {
                printk(KERN_ALERT "Failed to register virtio driver.\n");
                goto out_with_chrdev;
        }

        debug("Leaving");
        return ret;

out_with_chrdev:
        debug("Leaving");
        crypto_chrdev_destroy();
out:
        return ret;
}

/**
 * The function that is called when our module is being removed.
 * Make sure to cleanup everything.
 **/
static void __exit fini(void)
{
        debug("Entering");
        crypto_chrdev_destroy();
        unregister_virtio_driver(&virtio_crypto);
        debug("Leaving");
}

module_init(init);
module_exit(fini);

MODULE_DEVICE_TABLE(virtio, id_table);
MODULE_DESCRIPTION("Virtio crypto driver");
MODULE_LICENSE("GPL");
```