Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

Εργαστήριο Λειτουργικών Συστημάτων 7ου εξαμήνου - Ροή Υ

3η Εργαστηριακή Άσκηση
Κρυπτογραφική συσκευή VirtIO για QEMU-KVM

Μαρκουλέσκου Έλενα - Μπιάνκα **03115126** **bianca_marc_96@yahoo.com**	**Τσαπατσάρης Παναγιώτης** **03115102** **panagiotistsap@gmail.com**

Ζητούμενο 1

Στο ζητούμενο 1 αυτό που έπρεπε να κάνουμε ήταν να υλοποιήσουμε ένα chat το οποίο θα επιτρέπει την επικοινωνία μεταξύ διεργασιών στο ίδιο φυσικό μηχάνημα μέσω UNIX sockets. Αρχικά καταστήσαμε δυνατή την επικοινωνία μεταξύ των διεργασιών δημιουργώντας τα sockets, από τη μεριά του server και από τη μεριά του client, όπως στο παράδειγμα που μας δόθηκε. Ο server μας κάνει bind στην πόρτα 35001 και ακούει και δέχεται (το πολύ 5 ταυτόχρονες) συνδέσεις. Ο client μετά τη δημιουργία του socket κάνει connect στον localhost σε αυτή την πόρτα.

Επειδή μας ενδιαφέρει η αμφίδρομη επικοινωνία, θα χρειαστεί και από τις δύο πλευρές της σύνδεσης να παρακολουθούμε και τους δύο peers και να αποφασίζουμε τι θα κάνουμε κάθε φορά που κάποιος στέλνει ή δέχεται μήνυμα. Για το λόγο αυτό, σε κάθε πλευρά θα ορίσουμε 2 file descriptors: ο ένας θα είναι ο standard input στον οποίο θα γράφει το μήνυμα ο χρήστης και ο άλλος ο fd που παίρνουμε από την υφιστάμενη σύνδεση (αυτός που μας γυρνάει η accept από την πλευρά του server και αυτός που μας γυρνάει η connect από την πλευρά του client) και από τον οποίο δεχόμαστε μυνήματα από την άλλη πλευρά.

Για να μπορέσουμε να τους διαχειριστούμε θα χρησιμοποιήσουμε τη συνάρτηση poll. Η poll δέχεται ως όρισμα έναν πίνακα με pollfds τα οποία σαν πεδία έχουν το fd και το events που δείχνει τι ‘είδους’ ενέργεια περιμένουμε. Στο πεδίο events βάζουμε το POLLIN που σημαίνει ότι περιμένουμε είσοδο δεδομένων. Ακόμα δέχεται σαν ορίσματα τον αριθμό των pollfds που στην περίπτωση μας είναι 2 και τον αριθμό ms που θα περιμένει πριν γίνει timeout. ΄Οταν καλείται θα επιστρέψει 0 αν δεν έιναι έτοιμος κανένας fd και έχει γίνει timeout ή έναν θετικό αριθμό ίσο με αυτών των fds που είναι έτοιμοι. Αν είναι θετικός ο αριθμός κοιτάμε ποιος fd έχει πεδίο revents ίσο με POLLIN (δηλαδή έχει δεχτεί δεδομένα) και διαβάζουμε από αυτόν.

Πιο συγκεκριμένα, παρακολουθούμε τους file descriptors με τον εξής τρόπο. Θέτουμε χρονόμετρο ίσο με 5000ms (δηλαδή 5sec) και αν κάποιος από τους fds έχει έτοιμα δεδομένα τότε γίνεται διακοπή (αλλιώς γίνεται timeout). Αν ο έτοιμος fd είναι το standart input, διαβάζουμε τα δεδομένα από αυτό μέσα σε ένα buffer και τα γράφουμε στο ‘απέναντι’ socket. Αν έχουμε δεχτεί δεδομένα τότε τα γράφουμε στο standard output για να εμφανιστούν στο χρήστη.

Παρακάτω έχουμε τους κώδικες για τον server και τον client

socket-server.c

```c
#include <sys/poll.h>
#include <stdint.h>
#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>
```

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

#include "socket-common.h"
#include <fcntl.h>
#include <sys/ioctl.h>
#include <sys/stat.h>
#include <crypto/cryptodev.h>

/* 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[])
{
    unsigned char buf[BUF_SIZE];
    //char newbuf[BUF_SIZE];
    char addrstr[INET_ADDRSTRLEN];
    int sd, newsd,ret,cfd,po;
    ssize_t n;
    socklen_t len;
    struct sockaddr_in sa;
    struct pollfd fds[2];
    unsigned char crypbuf[BUF_SIZE];

    if (argc != 2) {
       fprintf(stderr, "Usage: %s key\n", argv[0]);
       exit(1);
    }
    /* Make sure a broken connection doesn't kill us */
    signal(SIGPIPE, SIG_IGN);
```

```
    /* Create TCP/IP socket, used as main chat channel */
    if ((sd = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
       perror("socket");
       exit(1);
    }
    fprintf(stderr, "Created TCP socket\n");

    /* Bind to a 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(sd, (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(sd, TCP_BACKLOG) < 0) {
       perror("listen");
       exit(1);
    }

    /* Loop forever, accept()ing connections */
    for (;;) {
       fprintf(stderr, "Waiting for an incoming connection...\n");

       /* Accept an incoming connection */
       len = sizeof(struct sockaddr_in);
       if ((newsd = accept(sd, (struct sockaddr *)&sa, &len)) < 0) {
             perror("accept");
             exit(1);
       }
       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));

       /* We break out of the loop when the remote peer goes away */

       for (;;) {
```

```
            fds[0].fd = 0;
            fds[0].events = POLLIN;
            //fds[0].revents = 0;
            fds[1].fd = newsd;
            fds[1].events = POLLIN;
            //continue;
            ret = poll(fds,2,5000);
            //printf("%d\n",ret);
            if (ret){
                  if(fds[0].revents == POLLIN){       //input apo plik
                        n = read(0,buf,sizeof(buf));
                        insist_write(newsd,buf,n);
                  }
                  if (fds[1].revents == POLLIN){      //input apo socket
                        n = read(newsd, buf, sizeof(buf));
                        if (n <= 0) {
                              if (n < 0)
                                    perror("read from remote peer
failed");
                              break;
                        }
                        else {
                              fprintf(stderr, "He said: ");
                              insist_write(1,buf,n);
                        }
                  }

            }
      }
   }
      /* Make sure we don't leak open files */
      if (close(newsd) < 0)
            perror("close");


   /* This will never happen */
   return 1;
}
```

socket-client.c

```
#include <sys/poll.h>
#include <stdint.h>
#include <stdio.h>
#include <errno.h>
#include <ctype.h>
```

```c
#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 "socket-common.h"
#include <fcntl.h>
#include <sys/ioctl.h>
#include <sys/stat.h>
#include <crypto/cryptodev.h>
#define TIMEOUT 1
#define    KEY_SIZE    16

/* 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 sd, port,po,cfd=-1;
    ssize_t n;
    unsigned char buf[BUF_SIZE];
    unsigned char crypbuf[BUF_SIZE];
    char *hostname;
    struct hostent *hp;
    struct sockaddr_in sa;
    int ret;
```

```
    struct pollfd fds[2];



    if (argc != 4) {
       fprintf(stderr, "Usage: %s hostname port key\n", argv[0]);
       exit(1);
    }
    hostname = argv[1];
    port = atoi(argv[2]); /* Needs better error checking */

    /* Create TCP/IP socket, used as main chat channel */
    if ((sd = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
       perror("socket");
       exit(1);
    }
    fprintf(stderr, "Created TCP socket\n");

    /* Look up remote hostname on DNS */
    if ( !(hp = gethostbyname(hostname))) {
       printf("DNS lookup failed for host %s\n", hostname);
       exit(1);
    }

    /* Connect to remote TCP port */
    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(sd, (struct sockaddr *) &sa, sizeof(sa)) < 0) {
       perror("connect");
       exit(1);
    }
    fprintf(stderr, "Connected.\n");



    fds[0].fd = 0;
    fds[0].events = POLLIN;

    fds[1].fd = sd;
    fds[1].events = POLLIN;

    /* Read answer and write it to standard output */
    for (;;) {
```

```
       ret = poll(fds, 2, 5000);
      if (ret){
            if (fds[1].revents == POLLIN){
                  n = read(sd, buf, sizeof(buf));
                  if (n<=0){
                        if (n < 0) {
                              perror("read");
                              exit(1);
                        }
                        else
                              fprintf(stderr,"Peer went away");
                        break;
                  }
                  fprintf(stderr, "He said: ");
                  insist_write(1,buf,n);


            }
            if (fds[0].revents == POLLIN){
                  n = read(0,buf,sizeof(buf));
                  insist_write(sd,buf,n);
            }
      }

    }

    fprintf(stderr, "\nDone.\n");
    return 0;
}
```

Ζητούμενο 2

Στο ζητούμενο 2 έπρεπε τα δεδομένα που στέλνουμε να είναι κρυπτογραφημένα και η άλλη πλευρά να τα αποκρυπτογραφεί. Αυτό γίνεται μέσω του module cryptodev.
Αρχικά ανοίγουμε το αρχείο “/dev/crypto” που είναι το ειδικό αρχείο της συσκευής cryptodev. Ορίζουμε κάποια κοινά κλειδιά για τον client και για τον server. Στη συνέχεια αρχικοποιούμε το session της κρυπτογράφησης και παίρνουμε το sesion id. Αφού διαβάσουμε το μήνυμα που θέλει να στείλει ο χρήστης το συμπληρώνουμε με τον χαρακτήρα ΄\0΄ μέχρι να γεμίσει ο buffer (έτσι ώστε το bufsize να είναι σύμφωνα με τις προδιαγραφές του cryptodev). Στην συνέχεια το κρυπτογραφούμε μέσω του cryptodev και το στέλνουμε απέναντι. Ο απέναντι ξανά μέσω του cryptodev τα αποκτυπτογραφεί και τα τυπώνει. Τα ΄\0΄ που έχουμε βάλει στο τέλος δεν τυπώνονται. Και πάλι για να είναι αμφίδρομη η επικοινωνία, δουλέψαμε όπως παραπάνω (ο παρακάτω κώδικας είναι επέκταση του κώδικα του 1ου ερωτήματος με

κατάλληλες τροποποιήσεις). Τέλος, όταν ένας από τους δύο peers κλείσει την επικοινωνία κλείνουμε το αντίστοιχο sesion και το “/dev/crypto”.
Συγκεκριμένα για την κρυπτογράφηση, έχουμε ορίσει τα structs cryp_op και session_op και αφού τους δώσαμε τις κατάλληλες πληροφορίες καλούμε την συνάρτηση ioctl με τα κατάλληλα ορίσματα. Για την αχρικοποίηση του sesison όριζουμε το κλειδί στο session_op session μας (και το keylen αντίστοιχα) και το cipher και καλούμε την ioctl(fd, CIOCGSESSION, &session). Αυτή μας γυρνάει το session id στο πεδίο ses του session. Για την κρυπτογράφηση και αποκρυπτογράφηση θα δώσουμε στο crypto_op crypt το αρχικοποιημένο πια session, το src (που είναι το κείμενο που θέλουμε να κρυπτογραφήσουμε), το iv και το op (encrypt ή decrypt). Θα καλέσουμε την ioctl(fd, CIOCCRYPT, crypt) και αυτή στο πεδίο dst του crypt θα μας δώσει το (κρυπτογραφημένο ή αποκρυπτογραφημένο) μήνυμα. Στα παραπάνω το fd είναι ο file descriptor που πήραμε όταν ανοίξαμε τον αρχείο της συσκευής cryptodev.
Παρακάρω δίνονται οι κώδικες για τον server και τον client:

socket-server.c

```c
#include <sys/poll.h>
#include <stdint.h>
#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 "socket-common.h"
#include <fcntl.h>
#include <sys/ioctl.h>
#include <sys/stat.h>
#include <crypto/cryptodev.h>

/* Convert a buffer to upercase */
void toupper_buf(char *buf, size_t n)
{
    size_t i;

    for (i = 0; i < n; i++)
       buf[i] = toupper(buf[i]);
```

```
}

/* 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[])
{
    unsigned char buf[BUF_SIZE];
    //char newbuf[BUF_SIZE];
    char addrstr[INET_ADDRSTRLEN];
    int sd, newsd,ret,cfd,po;
    ssize_t n;
    socklen_t len;
    struct sockaddr_in sa;
    struct pollfd fds[2];
    unsigned char crypbuf[BUF_SIZE];

    if (argc != 2) {
       fprintf(stderr, "Usage: %s key\n", argv[0]);
       exit(1);
    }
    /* Make sure a broken connection doesn't kill us */
    signal(SIGPIPE, SIG_IGN);

    /* Create TCP/IP socket, used as main chat channel */
    if ((sd = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
       perror("socket");
       exit(1);
    }
    fprintf(stderr, "Created TCP socket\n");
```

```
    /* Bind to a 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(sd, (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(sd, TCP_BACKLOG) < 0) {
       perror("listen");
       exit(1);
    }

    /* Loop forever, accept()ing connections */
    for (;;) {
       fprintf(stderr, "Waiting for an incoming connection...\n");

       /* Accept an incoming connection */
       len = sizeof(struct sockaddr_in);
       if ((newsd = accept(sd, (struct sockaddr *)&sa, &len)) < 0) {
             perror("accept");
             exit(1);
       }
       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));

       /* We break out of the loop when the remote peer goes away */

       //CRYPTO
       cfd = open("/dev/crypto", O_RDWR,0);
       if (cfd < 0) {
             perror("open(/dev/crypto)");
             return 1;
       }
       unsigned char iv[BLOCK_SIZE]="abcdefghabcdefgh";
```

```
    unsigned char key[KEY_SIZE] = "abdcabcdabcdabcd";
    //memcpy(key, argv[1], 16);
    struct session_op sess_op;

    memset(&sess_op,0,sizeof(sess_op));
    sess_op.key = key;
    sess_op.cipher = CRYPTO_AES_CBC;
    sess_op.keylen = KEY_SIZE;


    struct crypt_op cryp;
    memset(&cryp, 0, sizeof(cryp));
    cryp.iv = (void*)iv;
    if (ioctl(cfd, CIOCGSESSION, &sess_op)) {
                perror("ioctl");
                exit(1);
    }
    cryp.ses = sess_op.ses;
    cryp.len = BUF_SIZE;
//CRYPTO

    //fds[1].revents = 0;
    for (;;) {
          fds[0].fd = 0;
          fds[0].events = POLLIN;
          //fds[0].revents = 0;
          fds[1].fd = newsd;
          fds[1].events = POLLIN;
          //continue;
          ret = poll(fds,2,5000);
          //printf("%d\n",ret);
          if (ret){
                if(fds[0].revents == POLLIN){       //input apo plik
                      n = read(0,buf,sizeof(buf));
                      //insist_write(newsd,buf,n);
                      for(po=n;po<BUF_SIZE;po++)
                            buf[po]='\0';
                      //encrypt
                      cryp.ses = sess_op.ses;
                      cryp.src = buf;
                      cryp.dst = crypbuf;
                      cryp.len = sizeof(buf);
                      cryp.iv = (void*)iv;
                      cryp.op = COP_ENCRYPT;
                      if (ioctl(cfd, CIOCCRYPT,&cryp)){
```

```
                                perror("ioctl");
                                exit(1);
                        }
                        insist_write(newsd,crypbuf,sizeof(crypbuf));
                }
                if (fds[1].revents == POLLIN){      //input apo socket
                        n = read(newsd, buf, sizeof(buf));
                        if (n <= 0) {
                                if (n < 0)
                                        perror("read from remote peer
failed");
                                break;
                        }
                        else {
                                //insist_write(1,buf,n);
                                //decrypt
                                cryp.ses = sess_op.ses;
                                cryp.src = buf;
                                cryp.dst = crypbuf;
                                cryp.iv = (void*)iv;
                                cryp.len = sizeof(buf);
                                cryp.op = COP_DECRYPT;
                                if (ioctl(cfd, CIOCCRYPT ,&cryp)){
                                        perror("ioctl(decrypt)");
                                        exit(1);
                                }
                                if (crypbuf[0]=='\0')
                                        break;
                                fprintf(stderr, "He said: ");
                                if (insist_write(1, crypbuf,
sizeof(crypbuf)) != sizeof(crypbuf)) {
                                        perror("write to remote peer
failed");
                                        break;
                                }

                        }
                }

        }
    }
       /* Make sure we don't leak open files */
       if (close(newsd) < 0)
             perror("close");
```

```
    /* This will never happen */
    return 1;
}
```

socket-client.c

```
#include <sys/poll.h>
#include <stdint.h>
#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 "socket-common.h"
#include <fcntl.h>
#include <sys/ioctl.h>
#include <sys/stat.h>
#include <crypto/cryptodev.h>
#define TIMEOUT 1
#define    KEY_SIZE    16

/* 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 sd, port,po,cfd=-1;
    ssize_t n;
    unsigned char buf[BUF_SIZE];
    unsigned char crypbuf[BUF_SIZE];
    char *hostname;
    struct hostent *hp;
    struct sockaddr_in sa;
    int ret;
    struct pollfd fds[2];



    if (argc != 4) {
       fprintf(stderr, "Usage: %s hostname port key\n", argv[0]);
       exit(1);
    }
    hostname = argv[1];
    port = atoi(argv[2]); /* Needs better error checking */

    /* Create TCP/IP socket, used as main chat channel */
    if ((sd = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
       perror("socket");
       exit(1);
    }
    fprintf(stderr, "Created TCP socket\n");

    /* Look up remote hostname on DNS */
    if ( !(hp = gethostbyname(hostname))) {
       printf("DNS lookup failed for host %s\n", hostname);
       exit(1);
    }

    /* Connect to remote TCP port */
    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(sd, (struct sockaddr *) &sa, sizeof(sa)) < 0) {
       perror("connect");
       exit(1);
```

```
    }
    fprintf(stderr, "Connected.\n");

    //CRYPTO
    cfd = open("/dev/crypto", O_RDWR,0);
    if (cfd < 0) {
       perror("open(/dev/crypto)");
       return 1;
    }


    unsigned char key[KEY_SIZE] = "abdcabcdabcdabcd";
    unsigned char iv[BLOCK_SIZE]="abcdefghabcdefgh";


    struct session_op sess_op;
    memset(&sess_op,0,sizeof(sess_op));
    sess_op.key = key;
    sess_op.cipher = CRYPTO_AES_CBC;
    sess_op.keylen = KEY_SIZE;


    struct crypt_op cryp;
    memset(&cryp, 0, sizeof(cryp));
    cryp.iv = (void*)iv;


    if (ioctl(cfd, CIOCGSESSION, &sess_op)) {
             perror("ioctl");
             exit(1);
    }
    cryp.ses = sess_op.ses;
    cryp.len = BUF_SIZE;
    //CRYPTO

    fds[0].fd = 0;
    fds[0].events = POLLIN;

    fds[1].fd = sd;
    fds[1].events = POLLIN;

    /* Read answer and write it to standard output */
    for (;;) {
       ret = poll(fds, 2, 5000);
      if (ret){
```

```
            if (fds[1].revents == POLLIN){
                  n = read(sd, buf, sizeof(buf));
                  if (n<=0){
                        if (n < 0) {
                              perror("read");
                              exit(1);
                        }
                        else
                              fprintf(stderr,"Peer went away");
                        break;
                  }
                  //insist_write(1,buf,n);
                  //decrypt
                  cryp.src = buf;
                  cryp.ses = sess_op.ses;
                  cryp.dst = crypbuf;
                  cryp.len = sizeof(buf);
                  cryp.iv = (void*)iv;
                  cryp.op = COP_DECRYPT;
                  if (ioctl(cfd, CIOCCRYPT ,&cryp)){
                        perror("ioctl");
                        exit(1);
                  }
                  fprintf(stderr, "He said: ");
                  if (insist_write(1, crypbuf,sizeof(crypbuf)) !=
sizeof(crypbuf)) {
                        perror("write");
                        exit(1);
                  }


            }
            if (fds[0].revents == POLLIN){
                  n = read(0,buf,sizeof(buf));
                  for(po=n;po<BUF_SIZE;po++)
                        buf[po]='\0';
                  //encrypt
                  //insist_write(sd,buf,n);
                  cryp.iv = (void*)iv;
                  cryp.ses = sess_op.ses;
                  cryp.src = buf;
                  cryp.dst = crypbuf;
                  cryp.len = sizeof(buf);
                  cryp.op = COP_ENCRYPT;
                  if (ioctl(cfd, CIOCCRYPT,&cryp)){
```

```
                        perror("ioctl");
                        exit(1);
                    }
                    insist_write(sd,crypbuf,sizeof(crypbuf));
                }
            }

        }

        fprintf(stderr, "\nDone.\n");
        return 0;
}
```

Ζητούμενο 3

Στο ζητούμενο αυτό μας ζητήθηκε να φτιάξουμε μια κρυπτογραφική μηχανή η οποία να δουλεύει σε εικονικό περιβάλλον, χρησιμοποιώντας όμως και τους πόρους της φυσικής μηχανής του host, ύστερα από άμεση επικοινωνία μαζί του. Πρόκειται δηλαδή για Paravirtualization. Για το λόγο αυτό, χρειάζεται να χωρίσουμε τον οδηγό μας σε δύο κομμάτια: ένα frontend που εκτελείται στο χώρο πυρήνα του guest μηχανήματος και ένα backend που εκτελείται στο χώρο χρήστη του host, μέσα στη διεργασία του QEMU (μεταβαίνοντας σε χώρο πυρήνα του host όταν εκτελεί system calls).
Για να πετύχουμε το παραπάνω θα χρειαστεί να στείλουμε τα δεδομένα στον host, αυτός να τα διεκπεραιώνει τη ζητούμενη ενέργεια (πχ. κρυπτογράφηση μέσω του cryptodev) και ο host να στείλει πίσω στον guest το αποτέλεσμα της ενέργειας.
Για το κομμάτι του frontend τροποποιήσαμε τα αρχείο crypto.h, crypto-module.c και crypto-chrdev.c.

- Αρχείο crypto.h:

Εδώ θα προσθέσουμε ένα ακόμα πεδίο στο struct crypto_device που υλοποιεί τη συσκευή μας. Συγκεκριμένα θα βάλουμε έναν **σημαφόρο** lock έτσι ώστε να δώσουμε τη δυνατότητα στη συσκευή μας να έχει τον έλεγχο σε κρίσιμα τμήματα. Χρησιμοποιήσαμε σημαφόρο αντί για spinlock, διότι, όπως θα δούμε παρακάτω, όλα τα κρίσιμα τμήματα εκτελούνται σε **process context** οπότε μπορούν και να κοιμηθούν.

- Αρχείο crypto-module.c

Εδώ με την εντολή sema_init(&crdev->lock, 1); αρχικοποιήσαμε το σημαφόρο μας με τιμή 1, μέσα στη συνάρτηση virtcons_probe().

- Αρχείο crypto-chrdev.c: σε αυτό το αρχείο βρίσκονται όλες οι κλήσεις συστήματος που πραγματοποιεί η συσκευή μας, όπως η open, η close και η ioctl. Υλοποιήσαμε τις παρακάτω:
    - crypto_chrdev_open() : Αυτή η συνάρτηση κάνει συσχέτιση της συσκευής με τον iminor του inode και στην συνέχεια δημιουργεί ένα crypto open file. Αχρικοποιούμε τα πεδία του και έπειτα παράγουμει 2 scatterlists, μία για το syscall type και μία για το fd του host. Παίρνουμε το lock (αφού μεταβαίνουμε στο κρίσιμο τμήμα που είναι η προώθηση των δεδομένων στον host και η

επεξεργασία τους από αυτόν) και τις προσθέτουμε στο virtqueue του crdev. Στέλνουμε το virtqueue στον host μέσω της virtqueue kick και στην συνέχεια περιμένουμε μέχρι η virtqueue_get_buf να μην επιστρέψει null (η vitrqueue_get_buf επιστρέφει NULL όσο ο host δεν έχει επεξεργαστεί τα ζητούμενα δεδομένα). Αν όλα έχουν πάει καλά από την πλευρά του host, τότε μόλις βγούμε από το while loop θα έχουμε στο πεδίο host_fd του crof το fd που επέστρεψε ο host από το άνοιγμα της συσκευής κρυπτογράφησης cryptodev.

- crypto_chdev_release() : Η συνάρτηση αυτή υλοποιεί την close. Στέλνουμε στο backend πάλι μέσω των virtqueues το fd που θέλουμε να κλείσουμε και αυτό το κλείνει. Χρησιμοποιούμε και πάλι συγχρονισμό για να προστατεύσουμε το virtqueue. Τέλος απελευθερώνουμε τη μνήμη που είχαμε δεσμεύει για τη συσκευή crdev αλλά και για το crof.
- crypto_chrdev_ioctl() : Σε αυτήν την συνάρτηση υλοποιούμε τις κλήσεις ioctl. Αρχικά βάζουμε στη vq το syscall_type, το host_fd και το cmd αφού αυτά θα χρειαστεί να τα δώσουμε στον host σε κάθε περίπτωση. Ανάλογα με το comand που έχει δώσει η διεργασία του χρήστη, υλοποιούμε και το κατάλληλο ioctl. Διακρίνουμε τις περιπτώσεις:
  - CIOCGSESSION: Εδώ αντιγράφουμε από τη διεργασία του χρήστη το session, το βάζουμε στη vitrqueue και το στέλνουμε στο backend, όπου θα γίνει το “πραγματικό” άνοιγμα του session. Προσέχουμε να αντιγράψουμε και να στείλουμε το key (που είναι pointer) ξεχωριστά, αφού με την copy_from_user() αντιγράφονται οι διευθύνσεις των pointers αλλά όχι και τα περιεχόμενά τους. Ζητάμε από το backend να μας γυρίσει το ανανεωμένο session, το οποίο με τη σειρά μας αντιγράφουμε στη διεργασία του user. Τέλος, απελευθερώνουμε τη μνήμη από όλους τους pointers που δε μας χρειάζονται πια (όπως η μνήμη που δεσμεύσαμε για το session και το key αλλά και για το syscall_type και το cmd για παράδειγμα).
  - CIOCFSESSION: Ομοίως, στέλνουμε στο backend, μαζί με τα υπόλοιπα, και το session id έτσι ώστε αυτό να κλείσει το ζητούμενο session. Μόλις επιστρέψουμε από τον host, αποδεσμεύουμε μνήμη όπως παραπάνω.
  - CIOCCRYPT: Αντιγράφουμε κατάλληλα το crypt (όπως κάναμε και για το session, προσέχοντας τα πεδία που είναι pointers) και το στέλνουμε μαζί με τα πεδία του στο backend. Από αυτό περιμένουμε να πάρουμε στη θέση dst το κρυπτογραφημένο/αποκρυπτογραφημένο μήνυμα το οποίο θα αντιγράψουμε στο χρήστη στη διεύθυνση usr. Η διεύθυνση αυτή είναι η διεύθυνση του crypt.dst που βρίσκεται στο χώρο διευθύνσεων της διεργασίας του χρήστη και την οποία έχουμε φροντίσει να κρατήσουμε όταν αντιγράψαμε από το χρήστη το crypt. Και πάλι αποδεσμεύουμε μνήμη κατάλληλα.

  Η συνάρτηση γυρνάει την τιμή host_ret_val, που είναι το return value που παίρνουμε ως μέρος της υλοποίησης του backend.

Όσον αφορά το backend θα τροποποιήσουμε το αρχείο virtio-cryptodev.c που βρίσκεται στον πηγαίο κώδικα του QEMU στο directory qemu/hw/char. Στο αρχείο αυτό υλοποιούμε τη συνάρτηση vq_handle_output() που καλείτα κάθε φορά που κάνουμε virtqueue kick. Παίρνουμε από τη virtqueue το πρώτο element που έχει σταλεί και με βάση το πρώτο στοιχείο out (elem->out_sg[0].iov_base) βλέπουμε το syscall type που ζητείται να κάνουμε. Διακρίνουμε τις περιπτώσεις:

- VIRTIO_CRYPTODEV_SYSCALL_TYPE_OPEN: Εδώ θα κάνουμε open τη συσκευή cryptodev στο φυσική μηχάνημα και θα πάρουμε ένα fd. Με host_fd=elem->in_sg[0].iov_base; θα στείλουμε αυτό το fd στο frontend και συγκεκριμένα ως πεδίο του crof (αφού έτσι έχουμε ορίσει την αντίστοιχη λίστα scattergather). Εδώ να τονίσουμε πως για να βγάλουμε ή να βάλουμε στοιχεία μέσω DMA (που μας επιτρέπει το virtqueue) θα πρέπει να τηρήσουμε την αντίστοιχη σειρά με την οποία τα έχουμε προσθέσει στον πίνακα sgs.
- VIRTIO_CRYPTODEV_SYSCALL_TYPE_CLOSE: Εδώ θα εξάγουμε ακόμα από τη virtqueue το host_fd και θα κλείσουμε το αντίστοιχο ανοιχτό αρχείο που αντιστοιχεί σε αυτό το fd. Δε θα στείλουμε τίποτα πίσω στο frontend.
- VIRTIO_CRYPTODEV_SYSCALL_TYPE_IOCTL: Προσέχοντας τη σειρά με την οποία έχουμε βάλει στη λίστα sgs όλα τα arguments που μας χρειάζονται τα εξάγουμε κατάλληλα και κάνουμε ioctl ανάλογα με το ποιο cmd έχουμε λάβει. Στο τέλος βάζουμε στη virtqueue τα δεδομένα που πρέπει να στείλουμε στο frontend (το input message, το session σε περίπτωση CIOCGSESSION ή το κρυπτογραφημένο/αποκρυπτογραφημένο μήνυμα σε περίπτωση CIOCCRYPT, και το return value που είναι 0 σε περίπτωση επιτυχούς εκτέλεσης), πάλι προσέχοντας τη σειρά με την οποία υπάρχουν στην sgs.

Τέλος, με τη virtqueue_push() στέλνουμε αυτά τα δεδομένα στο frontend για να τα διαχειριστεί κατάλληλα.

Οι κώδικες για τα αρχεία crypto-chrdev.c και virtio-cryptodev.c φαίνονται παρακάτω:

GUEST

crypto-chrdev.c

```
/*
 * 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>
```

```c
#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;
    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 int num_out, num_in;

    num_out = 0;
    num_in = 0;
    debug("Entering");

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


    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.
     **/
    /* ?? */
    //initializing sg lists
    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;

    down(&crdev->lock);
    //adding virtqueue
    err = virtqueue_add_sgs(crdev->vq, sgs, num_out/*readable*/,
num_in/*writable*/, &syscall_type_sg, GFP_ATOMIC); //we don't want to
sleep
    if(err<0){
       debug("Didn't add to Virtqueue\n");
       ret = err;
       goto fail;
    }

    //kicking to host
    virtqueue_kick(crdev->vq);

    /**
     * Wait for the host to process our data.
     **/
    /* ?? */
    while(virtqueue_get_buf(crdev->vq, &len) == NULL) //returns null if
there are no used buffers
       ; /*do nothing*/
    up(&crdev->lock);
    /* If host failed to open() return -ENODEV. */
    /* ?? */
    //crof->host_fd=host_fd;
    debug("Host fd is %d\n", crof->host_fd);

    if(crof->host_fd<=0){
       debug("Failed to open\n");
       ret= -ENODEV;
    }

fail:
    kfree(syscall_type);
    debug("Leaving");
```

```
    return ret;
}

static int crypto_chrdev_release(struct inode *inode, struct file *filp)
{
    int ret = 0, err;
    struct crypto_open_file *crof = filp->private_data;
    struct crypto_device *crdev = crof->crdev;
    unsigned int *syscall_type;

    struct scatterlist syscall_type_sg, host_fd_sg, *sgs[2];
    unsigned int num_out, len;

    debug("Entering");

    syscall_type = kzalloc(sizeof(*syscall_type), GFP_KERNEL);
    *syscall_type = VIRTIO_CRYPTODEV_SYSCALL_CLOSE;
    /**
     * Send data to the host.
     **/
    /* ?? */
    num_out=0;

    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;

    down(&crdev->lock);
    err = virtqueue_add_sgs(crdev->vq, sgs, num_out/*readable*/, 0
/*nothing to write*/, &syscall_type_sg, GFP_ATOMIC); //we don't want to
sleep
    if(err<0){
       debug("Didn't add to Virtqueue\n");
       ret = err;
    }

    //kicking to host
    virtqueue_kick(crdev->vq);

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

```
    while(virtqueue_get_buf(crdev->vq, &len) == NULL) //returns null if
there are no used buffers
        ; //wait

    up(&crdev->lock);
    kfree(crdev);
    kfree(syscall_type);
    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;
    int *host_ret_val;
    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, cmd_sg, session_sg, key_sg, ses_id_sg, host_ret_val_sg,
crypt_sg, src_sg, iv_sg, dst_sg,
                    *sgs[13];
    unsigned int num_out, num_in, len;
#define MSG_LEN 100
    unsigned char *output_msg, *input_msg;
    unsigned int *syscall_type; // *host_fd;

    struct session_op *session;
    unsigned char *key, *src, *iv, *dst, *usr;
    struct crypt_op *crypt;
    uint32_t *ses_id;
    unsigned int *iocmd;

    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;
    iocmd = kzalloc(sizeof(*iocmd), GFP_KERNEL);
    *iocmd = cmd;
    host_ret_val = NULL;
    usr = NULL;
    dst = NULL;
    crypt = NULL;
    session = NULL;
    ses_id = NULL;
    src = NULL;
    iv = NULL;
    key = NULL;
    /**
     *  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;

    sg_init_one(&cmd_sg, iocmd, sizeof(*iocmd));
    sgs[num_out++] = &cmd_sg;

    /**
     *  Add all the cmd specific sg lists.
     **/
    switch (cmd) {
       case CIOCGSESSION:
             debug("CIOCGSESSION");
             memcpy(output_msg, "Hello HOST from ioctl CIOCGSESSION.",
36);
             input_msg[0] = '\0';
             sg_init_one(&output_msg_sg, output_msg, MSG_LEN);
             sgs[num_out++] = &output_msg_sg;
             session = kzalloc(sizeof(*session), GFP_KERNEL);
             if(copy_from_user(session, (struct session_op *)arg,
sizeof(*session))){
                   debug("Failed to copy session\n");
                   ret = -1;
```

```
                goto fail;
            }

            key = kzalloc(session->keylen*sizeof(unsigned char),
GFP_KERNEL);
            if(copy_from_user(key, session->key,
session->keylen*sizeof(unsigned char))){
                debug("Failed to copy key\n");
                ret = -1;
                goto fail;
            }

            debug("key is %s\n", key);
            sg_init_one(&key_sg, key, sizeof(*key));
            sgs[num_out++] = &key_sg;

            sg_init_one(&input_msg_sg, input_msg, MSG_LEN);
            sgs[num_out + num_in++] = &input_msg_sg;

            sg_init_one(&session_sg, session, sizeof(*session));
            sgs[num_out + num_in++] = &session_sg;

            host_ret_val = kzalloc(sizeof(*host_ret_val), GFP_KERNEL);
            sg_init_one(&host_ret_val_sg, host_ret_val,
sizeof(*host_ret_val));
            sgs[num_out + num_in++] = &host_ret_val_sg;


            break;

       case CIOCFSESSION:
            debug("CIOCFSESSION");
            memcpy(output_msg, "Hello HOST from ioctl CIOCFSESSION.",
36);
            input_msg[0] = '\0';
            sg_init_one(&output_msg_sg, output_msg, MSG_LEN);
            sgs[num_out++] = &output_msg_sg;

            ses_id = kzalloc(sizeof(*ses_id), GFP_KERNEL);
            if(copy_from_user(ses_id, (uint32_t*)arg, (sizeof
(uint32_t)))){
                debug("Failed to copy session id\n");
                ret=-1;
                goto fail;
            }
```

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

            sg_init_one(&input_msg_sg, input_msg, MSG_LEN);
            sgs[num_out + num_in++] = &input_msg_sg;

            host_ret_val = kzalloc(sizeof(*host_ret_val), GFP_KERNEL);
            sg_init_one(&host_ret_val_sg, host_ret_val,
sizeof(*host_ret_val));
            sgs[num_out + num_in++] = &host_ret_val_sg;

            break;

      case CIOCCRYPT:
            debug("CIOCCRYPT");
            memcpy(output_msg, "Hello HOST from ioctl CIOCCRYPT.", 33);
            input_msg[0] = '\0';
            sg_init_one(&output_msg_sg, output_msg, MSG_LEN);
            sgs[num_out++] = &output_msg_sg;

            crypt = kzalloc(sizeof(*crypt), GFP_KERNEL);
            if(copy_from_user(crypt, (struct crypt_op *)arg,
sizeof(*crypt))){
                  debug("Failed to copy crypt from user\n");
                  ret = 1;
                  goto fail;
            }
            sg_init_one(&crypt_sg, crypt, sizeof(*crypt));
            sgs[num_out++] = &crypt_sg;

            src = kzalloc(crypt->len*sizeof(char), GFP_KERNEL);
            if(copy_from_user(src, crypt->src,
crypt->len*sizeof(char))){
                  debug("Failed to copy source\n");
                  ret = 1;
                  goto fail;
            }
            sg_init_one(&src_sg, src, crypt->len*sizeof(char));
            sgs[num_out++] = &src_sg;

            iv = kzalloc(sizeof(__u8), GFP_KERNEL);
            if(copy_from_user(iv, crypt->iv, sizeof(__u8))){
                  debug("Failed to copy source\n");
                  ret = 1;
```

```
                goto fail;
            }
            sg_init_one(&iv_sg, iv, sizeof(__u8));
            sgs[num_out++] = &iv_sg;

            sg_init_one(&input_msg_sg, input_msg, MSG_LEN);
            sgs[num_out + num_in++] = &input_msg_sg;

            host_ret_val = kzalloc(sizeof(*host_ret_val), GFP_KERNEL);
           sg_init_one(&host_ret_val_sg, host_ret_val,
sizeof(*host_ret_val));
           sgs[num_out + num_in++] = &host_ret_val_sg;


            usr = crypt->dst; //save address for later
            dst = kzalloc(crypt->len*sizeof(char), GFP_KERNEL);

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


            break;

       default:
            debug("Unsupported ioctl command");

            break;
    }


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

    down(&crdev->lock);

    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 */;

    up(&crdev->lock);
```

```
    switch (cmd) {
       case CIOCGSESSION:
             debug("CIOGSESSION return");

             if(copy_to_user((struct session_op *)arg, session,
sizeof(struct session_op))){
                   debug("Failed to copy session id to user\n");
                   ret = -1;
                   goto fail;
             }
             kfree(key);
             kfree(session);
             break;

       case CIOCFSESSION:
             debug("CIOCFSESSION return");
             kfree(ses_id);
             break;

       case CIOCCRYPT:
             debug("CIOCCRYPT return");
             if(copy_to_user(usr, dst, crypt->len*sizeof(char))){
                   debug("Failed to copy dst to user\n");
                   ret = -1;
                   goto fail;
             }
             kfree(crypt);
             kfree(src);
             kfree(iv);
             break;

    }
    ret = *host_ret_val;

    debug("We said: '%s'", output_msg);
    debug("Host answered: '%s'", input_msg);

    kfree(iocmd);
    kfree(host_ret_val);
    kfree(output_msg);
    kfree(input_msg);
    kfree(syscall_type);

fail:
```

```
    debug("Leaving");
    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");
}
```

QEMU
hw/char/virtio-cryptodev.c

```
/*
 * 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>
```

```c
#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;
    int *host_fd;

    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");
             /* ?? */
             host_fd = elem->in_sg[0].iov_base;
             *host_fd = open("/dev/crypto", O_RDWR);
             printf("Host fd is %d\n", *host_fd);
             if(*host_fd<0){
                   perror("Did not open /dev/crypto\n");
                   return;
             }
             break;

       case VIRTIO_CRYPTODEV_SYSCALL_TYPE_CLOSE:
             DEBUG("VIRTIO_CRYPTODEV_SYSCALL_TYPE_CLOSE");
             /* ?? */
             host_fd = elem->out_sg[1].iov_base;
             if(close(*host_fd)<0){
                   perror("Could not close open file\n");
                   return;
             }
             break;

       case VIRTIO_CRYPTODEV_SYSCALL_TYPE_IOCTL:
             DEBUG("VIRTIO_CRYPTODEV_SYSCALL_TYPE_IOCTL");
             /* ?? */
             unsigned char *output_msg = elem->out_sg[3].iov_base;
             unsigned char *input_msg = elem->in_sg[0].iov_base;
            unsigned int *cmd;
            struct session_op *sess;
            struct crypt_op *crypt;
            unsigned char *key, *src, *dst, *iv;
            uint32_t *sess_id;
            int *ret;


             memcpy(input_msg, "Host: Welcome to the virtio World!",
35);
             printf("Guest says: %s\n", output_msg);
             printf("We say: %s\n", input_msg);

             host_fd = elem->out_sg[1].iov_base;
             cmd = elem->out_sg[2].iov_base;
```

```
        switch(*cmd) {
                case CIOCGSESSION:

                        sess = elem->in_sg[1].iov_base;
                        sess->key = elem->out_sg[4].iov_base;

                        ret = elem->in_sg[2].iov_base;
                        *ret = 0;
                        if(ioctl(*host_fd, CIOCGSESSION, sess)) {
                                perror("ioctl(CIOCGSESSION\n)");
                                *ret = 1;
                        }
                        break;

                case CIOCFSESSION:
                        sess_id = elem->out_sg[4].iov_base;
                        ret = elem->in_sg[1].iov_base;
                        *ret = 0;
                        if(ioctl(*host_fd, CIOCFSESSION, sess_id)){
                                perror("ioctl(CIOCFSESSION\n)");
                                *ret = 1;
                        }

                        break;
                case CIOCCRYPT:
                        crypt = elem->out_sg[4].iov_base;
                        src = elem->out_sg[5].iov_base;
                        crypt->src = src;

                        iv = elem->out_sg[6].iov_base;
                        crypt->iv = iv;

                        ret = elem->in_sg[1].iov_base;
                        *ret = 0;

                        dst = elem->in_sg[2].iov_base;
                        crypt->dst = dst;

                        if(ioctl(*host_fd, CIOCCRYPT, crypt)){
                                perror("ioctl(CIOCCRYPT\n)");
                                *ret = 1;
                        }

                        break;
```

```
            }

            break;

        default:
            DEBUG("Unknown syscall_type");
            break;
    }
    printf("pushing element to vq\n");
    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)
```