
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών Ε.Μ.Π.

ΕΡΓΑΣΤΗΡΙΟ ΛΕΙΤΟΥΡΓΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ
2015-2016

# ΑΣΚΗΣΗ 3η
# Κρυπτογραφική συσκευή VirtIO για QEMU-KVM

Χρυσούλα Βαρηά
ΑΜ: 03112105
Ευαγγελία-Σοφία Γεργατσούλη
ΑΜ: 03112064
Εξάμηνο:8ο

# 1 Ζητήματα 1 και 2

Στο chat που υλοποιήθηκε στη συγκεκριμένη άσκηση, γράφτηκαν 2 προγράμματα για τα δύο άκρα της επικοινωνίας: ένας server, ο οποίος θα δέχεται συνδέσεις, και ένας client, ο οποίος συνδέεται στον server και ανταλλάσσει μηνύματα με αυτόν. Οι δύο πλευρές της επικοινωνίας αναλύονται παρακάτω.

## 1.1 Chat: η πλευρά του server

Αρχικά ο server κάνει bind μια port (στην περίπτωση μας την 35001) στην οποία θα ακούει για εισερχόμενες συνδέσεις. Σε περίπτωση που ο client προσπαθήσει να συνδεθεί σε άλλη port, δεν θα τα καταφέρει, καθώς δεν θα πάρει απάντηση από τον server. Σημειώνεται ότι και ο client θα μπορούσε να κάνει bind μια port πριν συνδεθεί με τον server ώστε να συνδέεται πάντα από την ίδια, στην περίπτωση μας απλώς επιλέγεται μια διαθέσιμη.
Στη συνέχεια, ο server δέχεται μια εισερχόμενη σύνδεση μέσω της accept και σε περίπτωση που δεν υπάρχει καμία, γίνεται block μέχρι να έρθει κάποια. Αφού γίνει η αποδοχή κάποιας εισερχόμενης σύνδεσης ξεκινάει η συνομιλία μεταξύ server και client μέχρι να τερματιστεί η σύνδεση μεταξύ τους.

Για την ανταλλαγή και την ειδοποιήση λήψης μηνυμάτων χρησιμοποιείται η select. Συγκεκριμένα η select δέχεται δύο ορίσματα με ομάδες από file descriptors και ενημερώνει όταν κάποιος από αυτους είναι έτοιμος για read ή για write (ανάλογα αν είναι στους write ή στους read descriptors). Επιπλέον, η select μπλοκάρει μέχρι να υπάρξουν δεδομένα είτε στο αντίστοιχο socket (ο χρήστης λαμβάνει δεδομένα) είτε στο terminal (όπου χρήστης γράφει κάτι) είτε αν εξαντληθεί το χρονικό όριο, το οποίο έχει τεθεί μέσω της δομής timeval σε 1 sec. Όταν συμβεί κάποιο από τα παραπάνω γεγονότα εξετάζεται η αντίστοιχη περίπτωση (ανάλογα με το ποιο socket είναι set):

- Αν το newsd είναι set σημαίνει ότι ο client έστειλε δεδομένα στο server, τα οποία ο δεύτερος πρέπει να επεξεργαστεί.
- Αν το 0 είναι set σημαίνει ότι ο server έγραψε κάτι μέσω terminal, το οποίο πρέπει να σταλεί στον client.

Ο κώδικας για τον server φαίνεται αναλυτικά στην ενότητα 3.1.

## 1.2 Chat: η πλευρά του client

Στις ζητούμενες εφαρμογές έχει γίνει η παραδοχή ότι ο server επικοινωνεί με ένα client κάθε φορά, ενώ οι υπόλοιποι clients βρίσκονται σε αναμονή μέχρι να εξυπηρετηθεί ο αρχικός client και να κλείσει τη σύνδεση.

Ο client στέλνει αίτημα σύνδεσης μέσω της connect, με την οποία γίνεται η αντιστοίχιση του sd socket με την διεύθυνση sa. Η διαδικασία ειδοποίησης του client για τη λήψη μηνυμάτων είναι αντίστοιχη με αυτή που περιγράφτηκε για τη μεριά του server. Επιπλέον, για την πιθανή λήψη του σήματος $SIGSTOP$ (Ctrl+C) από τον χρήστη, υπάρχει αντίστοιχος signal handler για τον επιτυχή (και ομαλό) τερματισμό της σύνδεσης από τη μεριά του client.
Σε κάθε περίπτωση, οι client και server μπορούν να είναι υλοποιημένοι με πολύ διαφορετικό τρόπο και να ανήκουν σε διαφορετικά λειτουργικά συστήματα. Το μόνο απαραίτητο για να επικοινωνήσουν είναι να τηρούν το κοινό πρωτόκολλο.

Ο κώδικας για τον client φαίνεται αναλυτικά στην ενότητα 3.2.

### 1.3 Προσθήκη κρυπτογράφησης

Χρησιμοποιούμε το module cryptodev του linux για να υλοποιηθεί η κρυπτογράφηση της επικοινωνίας client-server μέσω κάποιων κλήσεων στη συσκευη /dev/crypto που παρουσιάζονται παρακάτω.
Αρχικά τα πεδία $key[], iv[]$ (key και initialization vector αντίστοιχα) της δομής data είναι προσυμφωνημένα και έχουν την ίδια τιμή τόσο από μεριά του client όσο και από αυτήν του server, χωρίς να χρειάζεται να επικοινωνήσουν για να τα καθορίσουν, κάτι που δε θα ήταν απόλυτα ασφαλές. Η έναρξη ενός session με τη συσκευή γίνεται με την αντίστοιχη κλήση ioctl (CIOCGSESSION) και από τις δύο πλευρές (από τη μεριά του server γίνεται μια κλήση για κάθε νέα σύνδεση, η οποία γίνεται accept). Αντίστοιχα ο τερματισμός του session γίνεται μέσω της κλήσης ioctl(CIOCFSESSION).
Η ενημέρωση για τη λήψη μηνυμάτων και από τις δύο πλευρές γίνεται με τη χρήση της select, όπως και στην επικοινωνία χωρίς κρυπτογράφηση. Σε αυτή την περίπτωση όμως, πριν την αποστολή κάποιου μηνύματος πρέπει να γίνει κρυπτογράφηση με τη χρήση ioctl(CIOCCRYPT) (με το flag COP_ENCRYPT ενεργοποιημένο), ενώ για την ανάγνωση ενός νέου μηνύματος απαιτείται αποκρυπτογράφηση, η οποία επιτυγχάνεται πάλι με την κλήση ioctl(CIOCCRYPT) αλλά με το flag COP_DECRYPT ενεργοποιημένο.
Σημειώνεται ότι σε αντίθεση με την πρώτη περίπτωση στην οποία γινόταν αποστολή μόνο του μηνύματος, εδώ γίνεται αποστολή όλου του περιεχομένου του buffer. Αυτό είναι απαραίτητο για τη σωστή διαδικασία της κρυπτογράφησης και αποκρυπτογράφησης, ενώ το περιεχόμενο του μηνύματος σταματάει με τον τερματικό χαρακτήρα $\backslash 0$ (NULL).

## 2 Ζήτημα 3ο

Στο ζήτημα 3 υλοποιήθηκε οδηγός για την κρυπτογραφική συσκευή που χρησιμοποιήσαμε στα πρώτα ερωτήματα. Συγκεκριμένα, θέλαμε ως χρήστης μέσα από μια εικονική μηχανή, να μπορούμε να χρησιμοποιούμε τις δυνατότητες της συσκευής κρυπτογράφησης cryptodev που υπάρχει ήδη στον host. Αυτό το πετυχαίνουμε με τη βοήθεια του virtio, με το οποίο μόνο οδηγός της συσκευής στον guest γνωρίζει ότι εκτελείται σε εικονικό περιβάλλον οπότε και συνεργάζεται κατάλληλα με τον hypervisor.

Η υλοποίηση μας υποστηρίζει 3 κλήσεις της ioctl για την συσκευή: τις CIOCGSESSION και CIOCFSESSION για τη δημιουργία και την καταστροφή ενός session αντίστοιχα, και την CIOCCRYPT για κρυπτογράφηση και αποκρυπτογράφηση. Η διαδικασία εκτέλεσης φαίνεται παρακάτω:

- Αρχικά, ο χρήστης της εικονικής μηχανής κάνει ένα system call (open, close, ioctl()) από το guest userspace.
- Για τον χειρισμό της κλήσης αυτής, γίνεται μετάβαση στον κώδικα frontend του qemu, όπου μέσω της copy_from_user() αντιγράφονται τα απαραίτητα δεδομένα από guest userspace σε guest kernel (με κατάλληλη μετάφραση διευθύνσεων στα 2 επίπεδα).
- Μόλις ολοκληρωθεί η αντιγραφή, μέσω της virtqueue_add_sgs() προστίθενται τα αντίστοιχα δεδομένα στην virtqueue.
- Έπειτα με τη χρήση της virtqueue_kick() ενημερώνεται ο host για την προσθήκη δεδομένων στη virtqueue. Μέσω αυτής της διαδικασίας μεταφράζονται οι διευθύνσεις από guest kernel σε physical space του host και ο έλεγχος μεταβαίνει στον κώδικα backend του qemu στο host userspace.
- Στη συνάρτηση vq_handle_output() στο αρχείο virtio-crypto.c γίνεται ο πραγματικός χειρισμός των αιτημάτων και η επικοινωνία με την συσκευη κρυπρογράφησης. Ο host λαμβάνει τα δεδομένα μέσω

της virtqueue_pop και εκτελεί τα αντίστοιχα system calls για την ικανοποίηση των αιτημάτων του guest user (κλήσεις από host userspace σε host kernel).

- Μόλις εκτελεστεί η αντίστοιχη κλήση, το αποτέλεσμα μεταβαίνει στο επίπεδο guest kernel και έπειτα σε guest userspace με χρήση της copy_to_user() αν κρίνεται απαραίτητο. Η ειδοποίηση των επεξεργασμένων δεδομένων από τον host γίνεται μέσω της virtqueue_push() και virtio_notify().

***Παρατηρήσεις-Σχόλια:*** Χρήση spinlocks:
Στον κώδικα του frontend χρησιμοποιούνται 2 locks, ένα για τον driver και ένα για κάθε συσκευή που αυτός χειρίζεται. Κάθε φορά που γίνεται open μιας συσκευής χρησιμοποιείται το crdrvdata.lock ώστε ο driver να χρησιμοποιείται κάθε φορά μόνο από μια συσκευή.
Όταν επίσης γίνεται εκτέλεση των virtqueue_add_sgs, virtqueue_kick και virtqueue_get_buf χρησιμοποιείται το crdev_lock (lock της κάθε συσκευής). Αυτό είναι απαραίτητο για να γνωρίζουμε τα δεδομένα που βρίσκονται στην virtqueue, σε ποιο αίτημα αντιστοιχούν (ποια διεργασία τα ζήτησε και ποια πρέπει να τα λάβει). Επισημαίνεται ότι μπορούν να χρησιμοποιηθούν σημαφόροι αντί για locks καθώς βρισκόμαστε σε proccess context, όμως επειδή η εκτέλεση των κλήσεων συστήματος είναι μικρής διάρκειας ίσως είναι λιγότερο χρονοβόρο να γίνεται polling αντί να μεταβαίνει η διεργασία σε waiting και έπειτα να χρειάζεται να ξυπνήσει.

Στην ενότητα 3.3 του παραρτήματος βρίσκεται ο πλήρης κώδικας για τον driver, ενώ στην ενότητα 3.4 παρατίθεται η συνάρτηση *handle_output* (από το αρχείο virtio-crypto.c) η οποία είναι υπεύθυνη για τους χειρισμούς από την πλευρά του qemu. στο αρχείο *crypto-module.c* το μόνο που προσθέσαμε είναι οι γραμμές που φαίνονται παρακάτω για την αρχικοποίηση του spinlocj στη συνάρτηση *static int virtcons_probe*:

```
1     /*Initialize the spin lock*/
2     spin_lock_init(&crdev->crdev_lock);
```

Τέλος στο αρχείο *crypto.h* τροποποιήθηκε η struct της συσκευης ως εξής, ώστε να περιέχει το 2ο spinlock:

```
1  struct crypto_device {
2     /* Next crypto device in the list, head is in the crdrvdata struct
          */
3     struct list_head list;
4
5     /* The virtio device we are associated with. */
6     struct virtio_device *vdev;
7
8     struct virtqueue *vq;
9     /*lock the device*/
10    spinlock_t crdev_lock;
11    /* The minor number of the device. */
12    unsigned int minor;
13 };
```

# 3 Παράρτημα: Κώδικας

## 3.1 Η πλευρά του server

```c
/*
 * socket-server.c
 * Simple TCP/IP communication using sockets
 *
 * Vangelis Koukis <vkoukis@cslab.ece.ntua.gr>
 */

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

#include <arpa/inet.h>
#include <netinet/in.h>
#include <crypto/cryptodev.h>
#include <sys/ioctl.h>
#include <sys/stat.h>
#include <fcntl.h>
#include "socket-common.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;
```

```
45 }
46
47 int main(void)
48 {
49     char buf[DATA_SIZE];
50     char addrstr[INET_ADDRSTRLEN];
51     int sd, newsd, maxsd, cfd, i;
52     int server_writes=0, server_reads=0;
53     ssize_t n;
54     socklen_t len;
55     fd_set read_fd_set;
56     struct sockaddr_in sa;
57     struct timeval tmv;
58     struct session_op sess;
59     struct crypt_op cryp;
60     struct {
61         unsigned char  in[DATA_SIZE],
62                encrypted[DATA_SIZE],
63                decrypted[DATA_SIZE],
64                iv[BLOCK_SIZE],
65                key[KEY_SIZE];
66     } data;
67
68     /* Make sure a broken connection doesn't kill us */
69     signal(SIGPIPE, SIG_IGN);
70
71     /* Create TCP/IP socket, used as main chat channel */
72     if ((sd = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
73         perror("socket");
74         exit(1);
75     }
76     fprintf(stderr, "Created␣TCP␣socket\n");
77
78     /* Bind to a well-known port */
79     memset(&sa, 0, sizeof(sa));
80     sa.sin_family = AF_INET;
81     sa.sin_port = htons(TCP_PORT);
82     sa.sin_addr.s_addr = htonl(INADDR_ANY);
83     if (bind(sd, (struct sockaddr *)&sa, sizeof(sa)) < 0) {
84         perror("bind");
85         exit(1);
86     }
87     fprintf(stderr, "Bound␣TCP␣socket␣to␣port␣%d\n", TCP_PORT);
88
89     /* Listen for incoming connections */
90     if (listen(sd, TCP_BACKLOG) < 0) {
91         perror("listen");
92         exit(1);
```

```
93      }
94
95
96
97      cfd = open("/dev/crypto", O_RDWR);
98      if (cfd < 0) {
99          perror("open(/dev/crypto)");
100         return 1;
101     }
102     /*Key must be the same for client and server. We agree on it
            before starting communicate*/
103     /*Same for initialization vector (iv)*/
104     for(i=0; i<KEY_SIZE; i++)
105         data.key[i]='x';
106     for(i=0; i<BLOCK_SIZE; i++)
107         data.iv[i]='y';
108
109     /* Loop forever, accept()ing connections */
110     for (;;) {
111         fprintf(stderr, "Waiting␣for␣an␣incoming␣connection...\n");
112
113         /* Accept an incoming connection */
114         len = sizeof(struct sockaddr_in);
115         if ((newsd = accept(sd, (struct sockaddr *)&sa, &len)) < 0) {
116             perror("accept");
117             exit(1);
118         }
119         if (!inet_ntop(AF_INET, &sa.sin_addr, addrstr, sizeof(addrstr))
                ) {
120             perror("could␣not␣format␣IP␣address");
121             exit(1);
122         }
123         fprintf(stderr, "Incoming␣connection␣from␣%s:%d\n",
124             addrstr, ntohs(sa.sin_port));
125
126
127             memset(&sess, 0, sizeof(sess));
128             memset(&cryp, 0, sizeof(cryp));
129             /*
130             * Get crypto session for AES128
131             */
132             sess.cipher = CRYPTO_AES_CBC;
133             sess.keylen = KEY_SIZE;
134             sess.key = data.key;
135
136             if (ioctl(cfd, CIOCGSESSION, &sess)) {
137                 perror("ioctl(CIOCGSESSION)");
138                 return 1;
```

```
139             }
140
141
142         for(;;){
143
144             tmv.tv_sec =1;
145             tmv.tv_usec=0;
146             FD_ZERO(&read_fd_set);
147                     FD_SET(0, &read_fd_set);
148             FD_SET(newsd,&read_fd_set);
149             maxsd=newsd;
150
151             if(select(maxsd+1, &read_fd_set,NULL, NULL, &tmv)<0){
152                     perror ("select");
153                     exit(1);
154                 }
155
156             /*Check if someone wrote something. It's either us or the
                   other client*/
157             if(FD_ISSET(0, &read_fd_set))
158                 server_writes=1;
159             else
160                 server_writes=0;
161
162             if(FD_ISSET(newsd, &read_fd_set))
163                 server_reads =1;
164             else
165                 server_reads =0;
166
167             /*In case we wrote something in stdin we need to send it*/
168             if(server_writes){
169
170                 n=read(0, buf, sizeof(buf));
171                 if (n < 0) {
172                     perror("Reading␣from␣command␣line");
173                     break;
174                 }
175                 if(n>0)
176                     buf[n-1]='\0';
177                 else
178                     buf[0]='\0';
179
180                 for(i=0;i<DATA_SIZE;i++)
181                     data.in[i] = buf[i];
182
183
184                 /*
185                 * Encrypt data.in to data.encrypted
```

```
186                 */
187                 cryp.ses = sess.ses;
188                 cryp.len = sizeof(data.in);
189                 cryp.src = data.in;
190                 cryp.dst = data.encrypted;
191                 cryp.iv = data.iv;
192                 cryp.op = COP_ENCRYPT;
193 
194                 if (ioctl(cfd, CIOCCRYPT, &cryp)) {
195                     perror("ioctl(CIOCCRYPT)");
196                     return 1;
197                 }
198 
199                 for(i=0;i<DATA_SIZE;i++)
200                     buf[i]=data.encrypted[i];
201 
202                 if (insist_write(newsd, data.encrypted, DATA_SIZE) !=
                        DATA_SIZE) {
203                     perror("write to remote peer failed");
204                     break;
205                 }
206 
207             }
208 
209             /*In case the other client wrote something we need to read
                    it*/
210             if(server_reads){
211 
212                 n=read(newsd, buf, DATA_SIZE);
213 
214                 if (n <= 0) {
215                     if (n < 0)
216                         perror("read from remote peer failed");
217                     else
218                         fprintf(stderr, "Peer went away\n");
219                     break;
220                 }
221 
222                 for(i=0;i<DATA_SIZE;i++)
223                     data.encrypted[i]=buf[i];
224 
225                 printf("Encrypted data:\n");
226                 for(i=0;i<DATA_SIZE;i++)
227                      printf("%x",data.encrypted[i]);
228                 printf("\n");
229 
230                 /*
231                 * Decrypt data.encrypted to data.decrypted
```

```
232                 */
233                 cryp.ses = sess.ses;
234                 cryp.len = sizeof(data.encrypted);
235                 cryp.src = data.encrypted;
236                 cryp.dst = data.decrypted;
237                 cryp.iv = data.iv;
238                 cryp.op = COP_DECRYPT;
239                 if (ioctl(cfd, CIOCCRYPT, &cryp)) {
240                     perror("ioctl(CIOCCRYPT)");
241                     return 1;
242                 }
243 
244 
245                 for(i=0;i<DATA_SIZE;i++)
246                     buf[i]=data.decrypted[i];
247 
248                 fprintf(stdout, "(Decrypted data)Client said:%s\n", buf);
249 
250             }
251 
252 
253 
254         }
255 
256         /* Make sure we don't leak open files */
257         if (close(newsd) < 0)
258             perror("close");
259 
260         /* Finish crypto session */
261         if (ioctl(cfd, CIOCFSESSION, &sess.ses)) {
262             perror("ioctl(CIOCFSESSION)");
263             return 1;
264         }
265     }
266 
267 
268     if (close(cfd) < 0) {
269         perror("close(fd)");
270         return 1;
271     }
272 
273     /* This will never happen */
274     return 1;
275 }
```

### 3.2 Η πλευρά του client

```
1  /*
2   * socket-client.c
```

```c
 * Simple TCP/IP communication using sockets
 *
 * Vangelis Koukis <vkoukis@cslab.ece.ntua.gr>
 */

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

#include <arpa/inet.h>
#include <netinet/in.h>
#include <crypto/cryptodev.h>
#include <sys/ioctl.h>
#include <sys/stat.h>
#include <fcntl.h>

#include "socket-common.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[])
{
    int sd, port, maxsd, cfd, i;
```

```
51      ssize_t n;
52      char buf[DATA_SIZE];
53      int client_writes=0, client_reads=0;
54      char *hostname;
55      struct hostent *hp;
56      struct sockaddr_in sa;
57      fd_set read_fd_set;
58      struct timeval tmv;
59      struct session_op sess;
60      struct crypt_op cryp;
61      struct {
62          unsigned char  in[DATA_SIZE],
63                 encrypted[DATA_SIZE],
64                 decrypted[DATA_SIZE],
65                 iv[BLOCK_SIZE],
66                 key[KEY_SIZE];
67      } data;
68
69
70      if (argc != 3) {
71          fprintf(stderr, "Usage: %s hostname port\n", argv[0]);
72          exit(1);
73      }
74      hostname = argv[1];
75      port = atoi(argv[2]);
76
77      if(port<=0 || port>65535){
78          fprintf(stderr, "%s is not a valid port number.\n", argv[2]);
79          exit(1);
80      }
81
82      /* Create TCP/IP socket, used as main chat channel */
83      if ((sd = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
84          perror("socket");
85          exit(1);
86      }
87      fprintf(stderr, "Created TCP socket\n");
88
89      /* Look up remote hostname on DNS */
90      if ( !(hp = gethostbyname(hostname))) {
91          printf("DNS lookup failed for host %s\n", hostname);
92          exit(1);
93      }
94
95      /* Connect to remote TCP port */
96      sa.sin_family = AF_INET;
97      sa.sin_port = htons(port);
98      memcpy(&sa.sin_addr.s_addr, hp->h_addr, sizeof(struct in_addr));
```

```
99      fprintf(stderr, "Connecting to remote host... "); fflush(stderr);
100
101     if (connect(sd, (struct sockaddr *) &sa, sizeof(sa)) < 0) {
102         perror("connect");
103         exit(1);
104     }
105     fprintf(stderr, "Connected.\n");
106     fprintf(stderr, "To end the connection press 'Ctrl+c'\n");
107
108     /*Key must be the same for client and server. We agree on it
            before starting communicate*/
109     /*Same for initialization vector (iv)*/
110     cfd = open("/dev/crypto", O_RDWR);
111     if (cfd < 0) {
112         perror("open(/dev/crypto)");
113         return 1;
114     }
115     for(i=0;i<KEY_SIZE;i++)
116         data.key[i]='x';
117     for(i=0; i<BLOCK_SIZE; i++)
118         data.iv[i]='y';
119
120     memset(&sess, 0, sizeof(sess));
121     memset(&cryp, 0, sizeof(cryp));
122
123     /*
124     * Get crypto session for AES128
125     */
126     sess.cipher = CRYPTO_AES_CBC;
127     sess.keylen = KEY_SIZE;
128     sess.key = data.key;
129
130     if (ioctl(cfd, CIOCGSESSION, &sess)) {
131         perror("ioctl(CIOCGSESSION)");
132         return 1;
133     }
134
135     for(;;){
136         tmv.tv_sec =1;
137         tmv.tv_usec=0;
138         FD_ZERO(&read_fd_set);
139               FD_SET(0, &read_fd_set);
140         FD_SET(sd,&read_fd_set);
141         maxsd=sd;
142
143         if(select(maxsd+1, &read_fd_set,NULL, NULL, &tmv)<0){
144                   perror ("select");
145                   exit(1);
```

```
146             }
147
148         /*Check if someone wrote something. It's either us or the
                other client*/
149         if(FD_ISSET(0, &read_fd_set))
150             client_writes=1;
151         else
152             client_writes=0;
153
154         if(FD_ISSET(sd, &read_fd_set))
155             client_reads =1;
156         else
157             client_reads =0;
158
159
160         /*In case we wrote something in stdin we need to send it*/
161         if(client_writes){
162
163             n=read(0, buf, sizeof(buf));
164             if (n < 0) {
165                 perror("Reading from command line");
166                 break;
167             }
168             if(n>0)
169                 buf[n-1]='\0';
170             else
171                 buf[0]='\0';
172
173             for(i=0;i<DATA_SIZE;i++)
174                 data.in[i] = buf[i];
175
176
177             /*
178             * Encrypt data.in to data.encrypted
179             */
180             cryp.ses = sess.ses;
181             cryp.len = sizeof(data.in);
182             cryp.src = data.in;
183             cryp.dst = data.encrypted;
184             cryp.iv = data.iv;
185             cryp.op = COP_ENCRYPT;
186
187             if (ioctl(cfd, CIOCCRYPT, &cryp)) {
188                 perror("ioctl(CIOCCRYPT)");
189                 return 1;
190             }
191
192             for(i=0;i<DATA_SIZE;i++)
```

```
193                 buf[i]=data.encrypted[i];
194
195             if (insist_write(sd, data.encrypted, DATA_SIZE) !=
                    DATA_SIZE) {
196                 perror("write to remote peer failed");
197                 break;
198             }
199
200         }
201
202         /*In case the other client wrote something we need to read
                it*/
203         if(client_reads){
204
205             n=read(sd, buf, DATA_SIZE);
206
207             if (n <= 0) {
208                 if (n < 0)
209                     perror("read from remote peer failed");
210                 else
211                     fprintf(stderr, "Peer went away\n");
212                 break;
213             }
214
215             for(i=0;i<DATA_SIZE;i++)
216                   data.encrypted[i]=buf[i];
217
218             printf("Encrypted data:\n");
219             for(i=0;i<DATA_SIZE;i++)
220                   printf("%x",data.encrypted[i]);
221             printf("\n");
222
223
224             /*
225             * Decrypt data.encrypted to data.decrypted
226             */
227             cryp.ses = sess.ses;
228             cryp.len = sizeof(data.encrypted);
229             cryp.src = data.encrypted;
230             cryp.dst = data.decrypted;
231             cryp.iv = data.iv;
232             cryp.op = COP_DECRYPT;
233             if (ioctl(cfd, CIOCCRYPT, &cryp)) {
234                 perror("ioctl(CIOCCRYPT)");
235                 return 1;
236             }
237
238
```

```
239                 for(i=0;i<DATA_SIZE;i++)
240                     buf[i]=data.decrypted[i];
241
242                 fprintf(stdout, "(Decrypted data)Server said:%s\n", buf);
243
244             }
245
246     }
247
248     /*
249      * Let the remote know we're not going to write anything else.
250      * Try removing the shutdown() call and see what happens.
251      */
252     printf("Client exiting...\n");
253
254     /* Finish crypto session */
255     if (ioctl(cfd, CIOCFSESSION, &sess.ses)) {
256         perror("ioctl(CIOCFSESSION)");
257         return 1;
258     }
259
260     if (close(cfd) < 0) {
261         perror("close(fd)");
262         return 1;
263     }
264
265     if (shutdown(sd, SHUT_WR) < 0) {
266         perror("shutdown");
267         exit(1);
268     }
269
270
271     fprintf(stderr, "\nDone.\n");
272     return 0;
273 }
```

### 3.3 O driver (crypto-chrdev.c)

```
1  /*
2   * crypto-chrdev.c
3   *
4   * Implementation of character devices
5   * for virtio-crypto device
6   *
7   * Vangelis Koukis <vkoukis@cslab.ece.ntua.gr>
8   * Dimitris Siakavaras <jimsiak@cslab.ece.ntua.gr>
9   * Stefanos Gerangelos <sgerag@cslab.ece.ntua.gr>
10  *
11  */
```

```
12 #include <linux/cdev.h>
13 #include <linux/poll.h>
14 #include <linux/sched.h>
15 #include <linux/module.h>
16 #include <linux/wait.h>
17 #include <linux/virtio.h>
18 #include <linux/virtio_config.h>
19
20 #include "crypto.h"
21 #include "crypto-chrdev.h"
22 #include "debug.h"
23
24 #include "cryptodev.h"
25
26 #define KEY_SIZE 24
27 #define DATA_SIZE        16384
28 /*
29  * Global data
30  */
31 struct cdev crypto_chrdev_cdev;
32
33
34 /**
35  * Given the minor number of the inode return the crypto device
36  * that owns that number.
37  **/
38 static struct crypto_device *get_crypto_dev_by_minor(unsigned int
      minor)
39 {
40     struct crypto_device *crdev;
41     unsigned long flags;
42
43     debug("Entering");
44
45     spin_lock_irqsave(&crdrvdata.lock, flags);
46     list_for_each_entry(crdev, &crdrvdata.devs, list) {
47         if (crdev->minor == minor)
48             goto out;
49     }
50     crdev = NULL;
51
52 out:
53     spin_unlock_irqrestore(&crdrvdata.lock, flags);
54
55     debug("Leaving");
56     return crdev;
57 }
58
```

```
59 /************************************
60  * Implementation of file operations
61  * for the Crypto character device
62  ************************************/
63
64 static int crypto_chrdev_open(struct inode *inode, struct file *filp)
65 {
66     int ret = 0;
67     int err;
68     unsigned long flags;
69     unsigned int len;
70     struct crypto_open_file *crof;
71     struct crypto_device *crdev;
72     unsigned int syscall_type = VIRTIO_CRYPTO_SYSCALL_OPEN;
73     int host_fd = -1;
74     struct scatterlist syscall_type_sg, host_fd_sg, *sgs[2];
75
76     debug("Entering");
77
78     ret = -ENODEV;
79     if ((ret = nonseekable_open(inode, filp)) < 0)
80         goto fail;
81
82     /* Associate this open file with the relevant crypto device. */
83     crdev = get_crypto_dev_by_minor(iminor(inode));
84     if (!crdev) {
85         debug("Could not find crypto device with %u minor",
86               iminor(inode));
87         ret = -ENODEV;
88         goto fail;
89     }
90
91     crof = kzalloc(sizeof(*crof), GFP_KERNEL);
92     if (!crof) {
93         ret = -ENOMEM;
94         goto fail;
95     }
96
97     crof->crdev = crdev;
98     crof->host_fd = -1;
99     filp->private_data = crof;
100    debug("kzalloc ok");
101    /**
102     * We need two sg lists, one for syscall_type and one to get the
103     * file descriptor from the host.
104     **/
105    sg_init_one(&syscall_type_sg, &syscall_type, sizeof(syscall_type))
        ;
```

```
106     sgs[0] = &syscall_type_sg;
107     sg_init_one(&host_fd_sg, &host_fd, sizeof(host_fd));
108     sgs[1] = &host_fd_sg;
109
110  spin_lock_irqsave(&crdrvdata.lock, flags);
111     err = virtqueue_add_sgs(crdev->vq, sgs, 1, 1, &syscall_type_sg,
            GFP_ATOMIC);
112
113     if(err<0){
114         debug("something wrong");
115         ret = -ENODEV;
116         goto fail;
117     }
118
119     virtqueue_kick(crdev->vq);
120     /**
121      * Wait for the host to process our data.
122      **/
123     while (virtqueue_get_buf(crdev->vq, &len) == NULL)
124         /*do nothing*/;
125
126     spin_unlock_irqrestore(&crdrvdata.lock, flags);
127     /* If host failed to open() return -ENODEV. */
128     if(host_fd <0){
129         ret = -ENODEV;
130         goto fail;
131     }
132
133     crof-> host_fd = host_fd;
134     printk("host_fd = %d\n", crof->host_fd);
135  fail:
136     debug("Leaving");
137     return ret;
138  }
139
140  static int crypto_chrdev_release(struct inode *inode, struct file *
        filp)
141  {
142     int ret = 0;
143     int err;
144     unsigned int len;
145     unsigned long flags;
146     struct crypto_open_file *crof = filp->private_data;
147     struct crypto_device *crdev = crof->crdev;
148     unsigned int syscall_type = VIRTIO_CRYPTO_SYSCALL_CLOSE;
149     struct scatterlist syscall_type_sg, host_fd_sg, *sgs[2];
150
151     debug("Entering");
```

```
152
153     /**
154      * Send data to the host.
155      **/
156     sg_init_one(&syscall_type_sg, &syscall_type, sizeof(syscall_type))
            ;
157     sgs[0] = &syscall_type_sg;
158     sg_init_one(&host_fd_sg, &(crof->host_fd), sizeof(crof->host_fd));
159     sgs[1] = &host_fd_sg;
160
161 spin_lock_irqsave(&crdrvdata.lock, flags);
162     err = virtqueue_add_sgs(crdev->vq, sgs, 2, 0,
163                             &syscall_type_sg, GFP_ATOMIC);
164
165     virtqueue_kick(crdev->vq);
166     /**
167      * Wait for the host to process our data.
168      **/
169     while (virtqueue_get_buf(crdev->vq, &len) == NULL)
170         /*do nothing*/;
171
172     spin_unlock_irqrestore(&crdrvdata.lock, flags);
173
174     kfree(crof);
175     debug("Leaving");
176     return ret;
177 }
178
179 static long crypto_chrdev_ioctl(struct file *filp, unsigned int cmd,
180                                 unsigned long arg)
181 {
182     long ret = 0;
183     int err;
184     unsigned long flags;
185     struct crypto_open_file *crof = filp->private_data;
186     struct crypto_device *crdev = crof->crdev;
187     struct virtqueue *vq = crdev->vq;
188     struct scatterlist syscall_type_sg, output_msg_sg, input_msg_sg,*
            sgs[8];
189     struct scatterlist host_fd_sg, ioctl_cmd_sg, session_key_sg,
            sess_sg, sess_id_sg, host_ret_sg, cryp_sg, src_sg, dst_sg, iv_sg
            ;
190     #define MSG_LEN 100
191
192     unsigned char output_msg[MSG_LEN], input_msg[MSG_LEN];
193     unsigned int num_out, num_in,
194                  syscall_type = VIRTIO_CRYPTO_SYSCALL_IOCTL,
195                  len;
```

```
196
197     /*Common fields of CIOXXSESSION*/
198     unsigned int ioctl_cmd=cmd;
199     int host_return_val;
200     /*Fields of CIOCGSESSION*/
201     unsigned char session_key[KEY_SIZE];
202     struct session_op sess;
203     struct session_op * temp;
204     /*Fields of CIOCFSESSION*/
205     u32 sess_id=crdev->minor;
206     /*Fields of CIOCCRYPT*/
207     struct crypt_op cryp;
208     struct crypt_op * temp2;
209     unsigned char *src=NULL;
210     unsigned char iv[BLOCK_SIZE];
211     unsigned char *dst=NULL;
212
213     int i=0;
214     debug("Entering");
215
216     num_out = 0;
217     num_in = 0;
218
219     /**
220      *  These are common to all ioctl commands.
221      **/
222     sg_init_one(&syscall_type_sg, &syscall_type, sizeof(syscall_type))
          ;
223     sgs[num_out++] = &syscall_type_sg;
224     sg_init_one(&host_fd_sg, &(crof->host_fd), sizeof((crof->host_fd))
          );
225     sgs[num_out++] = &host_fd_sg;
226     sg_init_one(&ioctl_cmd_sg, &ioctl_cmd, sizeof(ioctl_cmd));
227     sgs[num_out++] = &ioctl_cmd_sg;
228
229     /**
230      *  Add all the cmd specific sg lists.
231      **/
232     switch (cmd) {
233     case CIOCGSESSION:
234         debug("CIOCGSESSION");
235         temp = (struct session_op __user * ) arg;
236
237         if(copy_from_user(&sess, temp, sizeof(*temp))){
238             ret = -EFAULT;
239         }
240         if(copy_from_user(session_key, (__u8 __user *)(temp->key), sess
              .keylen*sizeof(unsigned char))){
```

```
241             ret = -EFAULT;
242         }
243
244         sg_init_one(&session_key_sg, session_key, sizeof(*session_key))
                ;
245         sgs[num_out++] = &session_key_sg;
246         sg_init_one(&sess_sg, &sess, sizeof(struct session_op));
247         sgs[num_out + num_in++] = &sess_sg;
248         break;
249
250     case CIOCFSESSION:
251         debug("CIOCFSESSION");
252         sg_init_one(&sess_id_sg, &sess_id, sizeof(sess_id));
253         sgs[num_out++] = &sess_id_sg;
254
255         break;
256
257     case CIOCCRYPT:
258         debug("CIOCCRYPT");
259         temp2 = (struct crypt_op __user *) arg;
260         if(copy_from_user(&cryp, temp2, sizeof(*temp2))){
261             ret = -EFAULT;
262         }
263
264         src = kzalloc(cryp.len*sizeof(unsigned char), GFP_KERNEL);
265         if (!src) {
266             ret = -ENOMEM;
267             goto fail;
268         }
269         dst = kzalloc(cryp.len*sizeof(unsigned char), GFP_KERNEL);
270         if (!dst) {
271             ret = -ENOMEM;
272             goto fail;
273         }
274         if(copy_from_user(src, (__u8 __user *)(temp2->src), cryp.len*
                sizeof(unsigned char))){
275             ret = -EFAULT;
276         }
277         if(copy_from_user(iv, (__u8 __user *)(temp2->iv), BLOCK_SIZE*
                sizeof(unsigned char))){
278             ret = -EFAULT;
279         }
280         if(copy_from_user(dst, (__u8 __user *)(temp2->dst), cryp.len*
                sizeof(unsigned char))){
281             ret = -EFAULT;
282         }
283         sg_init_one(&cryp_sg, &cryp, sizeof(struct crypt_op));
284         sgs[num_out++] = &cryp_sg;
```

```
285         sg_init_one(&src_sg, src, sizeof(*src));
286         sgs[num_out++] = &src_sg;
287         sg_init_one(&iv_sg, iv, sizeof(*iv));
288         sgs[num_out++] = &iv_sg;
289         sg_init_one(&dst_sg, dst, sizeof(*dst));
290         sgs[num_out + num_in++] = &dst_sg;
291         break;
292     default:
293         debug("Unsupported ioctl command");
294         break;
295     }
296 
297     sg_init_one(&host_ret_sg, &host_return_val, sizeof(host_return_val
        ));
298     sgs[num_out + num_in++] = &host_ret_sg;
299 
300     /**
301      * Wait for the host to process our data.
302      **/
303     spin_lock_irqsave(&crdev->crdev_lock, flags);
304     err = virtqueue_add_sgs(crdev->vq, sgs, num_out, num_in,
305                             &syscall_type_sg, GFP_ATOMIC);
306 
307     virtqueue_kick(crdev->vq);
308     while (virtqueue_get_buf(crdev->vq, &len) == NULL)
309         ;
310 
311     spin_unlock_irqrestore(&crdev->crdev_lock, flags);
312     switch(cmd){
313         case CIOCCRYPT:
314             temp2 = (struct crypt_op __user * ) arg;
315             if(copy_to_user((__u8 __user *)(temp2->dst), dst, cryp.len*
                    sizeof(unsigned char))){
316                 ret = -EFAULT;
317             }
318             break;
319         case CIOCGSESSION:
320             temp = (struct session_op __user * ) arg;
321             if(copy_to_user(temp, &sess, sizeof(*temp))){
322                 ret = -EFAULT;
323             }
324             break;
325     }
326 
327     debug("Leaving");
328 fail:
329     return ret;
330 }
```

```
331
332 static ssize_t crypto_chrdev_read(struct file *filp, char __user *
        usrbuf,
333                                    size_t cnt, loff_t *f_pos)
334 {
335     debug("Entering");
336     debug("Leaving");
337     return -EINVAL;
338 }
339
340 static struct file_operations crypto_chrdev_fops =
341 {
342     .owner          = THIS_MODULE,
343     .open           = crypto_chrdev_open,
344     .release        = crypto_chrdev_release,
345     .read           = crypto_chrdev_read,
346     .unlocked_ioctl = crypto_chrdev_ioctl,
347 };
348
349 int crypto_chrdev_init(void)
350 {
351     int ret;
352     dev_t dev_no;
353     unsigned int crypto_minor_cnt = CRYPTO_NR_DEVICES;
354
355     debug("Initializing character device...");
356     cdev_init(&crypto_chrdev_cdev, &crypto_chrdev_fops);
357     crypto_chrdev_cdev.owner = THIS_MODULE;
358
359     dev_no = MKDEV(CRYPTO_CHRDEV_MAJOR, 0);
360     ret = register_chrdev_region(dev_no, crypto_minor_cnt, "
        crypto_devs");
361     if (ret < 0) {
362         debug("failed to register region, ret = %d", ret);
363         goto out;
364     }
365     ret = cdev_add(&crypto_chrdev_cdev, dev_no, crypto_minor_cnt);
366     if (ret < 0) {
367         debug("failed to add character device");
368         goto out_with_chrdev_region;
369     }
370
371     debug("Completed successfully");
372     return 0;
373
374 out_with_chrdev_region:
375     unregister_chrdev_region(dev_no, crypto_minor_cnt);
376 out:
```

```
377     return ret;
378 }
379
380 void crypto_chrdev_destroy(void)
381 {
382     dev_t dev_no;
383     unsigned int crypto_minor_cnt = CRYPTO_NR_DEVICES;
384
385     debug("entering");
386     dev_no = MKDEV(CRYPTO_CHRDEV_MAJOR, 0);
387     cdev_del(&crypto_chrdev_cdev);
388     unregister_chrdev_region(dev_no, crypto_minor_cnt);
389     debug("leaving");
390 }
```

### 3.4 Η συνάρτηση handle_output

```
1  static void vq_handle_output(VirtIODevice *vdev, VirtQueue *vq)
2  {
3      VirtQueueElement elem;
4      unsigned int *syscall_type;
5
6      DEBUG_IN();
7
8      if (!virtqueue_pop(vq, &elem)) {
9          DEBUG("No item to pop from VQ :(");
10         return;
11     }
12
13     DEBUG("I have got an item from VQ :)");
14
15     syscall_type = elem.out_sg[0].iov_base;
16     switch (*syscall_type) {
17     case VIRTIO_CRYPTO_SYSCALL_TYPE_OPEN:
18         DEBUG("VIRTIO_CRYPTO_SYSCALL_TYPE_OPEN");
19         int *h_fd = elem.in_sg[0].iov_base;
20         *h_fd = open("/dev/crypto", O_RDWR);
21         if (*h_fd < 0) {
22             perror("open(/dev/crypto)");
23             *h_fd=-2;
24         }
25         DEBUG("DONE WITH IT");
26         break;
27
28     case VIRTIO_CRYPTO_SYSCALL_TYPE_CLOSE:
29         DEBUG("VIRTIO_CRYPTO_SYSCALL_TYPE_CLOSE");
30         int *fd = elem.out_sg[1].iov_base;
31         if(close(*fd)<0){
32             perror("close");
```

```
33          }
34          break;
35
36      case VIRTIO_CRYPTO_SYSCALL_TYPE_IOCTL:
37          DEBUG("VIRTIO_CRYPTO_SYSCALL_TYPE_IOCTL");
38
39          unsigned int *cmd = elem.out_sg[2].iov_base;
40          int *host_return_val;
41          int *host_fd = elem.out_sg[1].iov_base;
42          printf("host_fd = %d\n", *host_fd);
43          switch (*cmd) {
44              case CIOCGSESSION:
45                  DEBUG("CIOCGSESSION");
46                  unsigned char *session_key=elem.out_sg[3].iov_base;
47                  struct session_op *sess=elem.in_sg[0].iov_base;
48                  host_return_val = elem.in_sg[1].iov_base;
49                  sess->key = session_key;
50                  if (ioctl((int)(*host_fd), CIOCGSESSION, sess)) {
51                      perror("ioctl(CIOCGSESSION)");
52                      *host_return_val=-1;
53                  }
54                  printf("after the ioctl\n");
55                  break;
56
57              case CIOCFSESSION:
58                  DEBUG("CIOCFSESSION");
59                  uint32 *ses_id = elem.out_sg[3].iov_base;
60                  host_return_val = elem.in_sg[0].iov_base;
61                  printf("end the crypto session\n");
62                  if (ioctl(*host_fd, CIOCFSESSION, ses_id)) {
63                              perror("ioctl(CIOCFSESSION)");
64                              *host_return_val=-1;
65                  }
66                  break;
67              case CIOCCRYPT:
68                  DEBUG("CIOCCRYPT");
69                  printf("in ioctl CIOCCRYPT\n");
70                  unsigned char *src = elem.out_sg[4].iov_base;
71                  unsigned char *iv = elem.out_sg[5].iov_base;
72                  unsigned char *dst = elem.in_sg[0].iov_base;
73                  struct crypt_op *cryp = elem.out_sg[3].iov_base;
74                  host_return_val = elem.in_sg[1].iov_base;
75
76                  cryp->src = src;
77                  cryp->iv = iv;
78                  cryp->dst = dst;
79
80                  if (ioctl(*host_fd, CIOCCRYPT, cryp)) {
```

```
81                      perror("ioctl(CIOCCRYPT)");
82                      *host_return_val=-1;
83                  }
84
85                  break;
86
87              default:
88                  DEBUG("Unsupported ioctl command");
89
90              break;
91          }
92          break;
93
94      default:
95          DEBUG("Unknown syscall_type");
96      }
97
98      virtqueue_push(vq, &elem, 0);
99      virtio_notify(vdev, vq);
100     DEBUG("DONE");
101 }
```