ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Εργαστήριο Λειτουργικών Συστημάτων

Κρυπτογραφική Συσκευή VirtIO για QEMU-KVM

7ο Εξάμηνο - Ακ. Έτος 2019-20

Ξενίας Δημήτριος
Α.Μ.: 03115084

Ειρήνη Γραφιαδέλλη
Α.Μ.: 03114112

1 Εισαγωγή

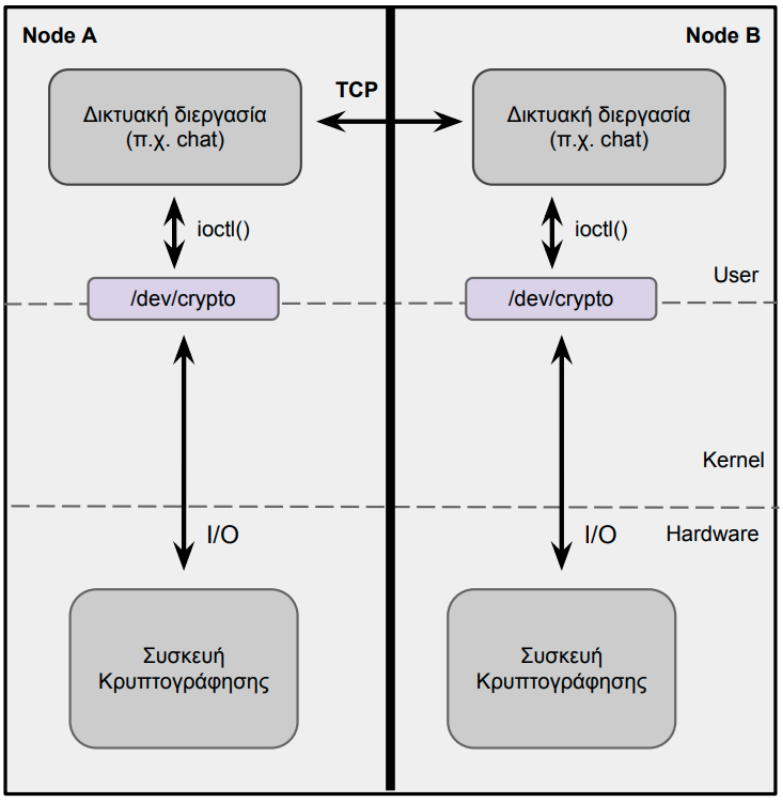
Στόχος της συγκεκριμένης εργαστηριακής άσκησης είναι η ανάπτυξη εικονικού *υλικού* στο περιβάλλον εικονικοποίησης QEMU-KVM. Στο πλαίσιο της συγκεκριμένης άσκησης σχεδιάσαμε και υλοποιήσαμε κρυπτογραφική συσκευή VirtIO, η οποία θα αποτελεί μέρος του QEMU. Η εικονική κρυπτογραφική συσκευή θα επιτρέπει σε διεργασίες που εκτελούνται μέσα στην εικονική μηχανή να έχουν πρόσβαση σε πραγματική κρυπτογραφική συσκευή του host, με χρήση τεχνικής *παρα-εικονοποίησης* (paravirtualization).

2 Κρυπτογραφημένη επικοινωνία πάνω από TCP/IP

Μέρος της άσκησης και πρώτο ζητούμενό της είναι η κατασκευή μιας εφαρμογής chat με την οποία επιτυγχάνεται κρυπτογραφημένη επικοινωνία. Τα δυο άκρα στα οποία εκτελείται η εφαρμογή επικοινωνούν μέσω της σουίτας TCP/IP, που είναι και ο πιο διαδεδομένος τρόπος επικοινωνίας στο Internet. Η υλοποίηση αυτής της επικοινωνίας γίνεται με χρήση του BSD Sockets API, ενώ η κρυπτογράφηση γίνεται μέσω του cryptodev userpace API (/dev/crypto)

Συγκεκριμένα η συσκευή cryptodev-linux επιτρέπει σε εφαρμογές να έχουν πρόσβαση σε επιταχυντές υλικού για κρυπτογραφία (hardware crypto accelerators), κάνοντας ioctl κλήσεις στο ειδικό αρχείο /dev/crypto.

Το σενάριο χρήσης της συγκεκριμένης εφαρμογής φαίνεται παρακάτω:



Σχήμα 1: Κρυπτογραφημένο chat πάνω από TCP/IP

2.1 Ζητούμενο 1: Εργαλείο chat πάνω από TCP/IP sockets

Αρχικά ας δούμε πως υλοποιούμε σε λογισμικό μια τέτοια επικοινωνία TCP/IP ανάμεσα σε δυο κόμβους του internet. Για το σκοπό αυτό δημιουργούμε έναν server και έναν client, στα πρότυπα της αρχιτεκτονικής client-server για την ανταλλαγή μηνυμάτων στο internet. Σενάριο χρήσης είναι ο client να επικοινωνεί μέσω chat με τον server χωρίς κρυπτογραφημένα μηνύματα (για απλούστευση αρχικά).

Για να πετύχουμε το σκοπό μας, χρησιμοποιούμε το BSD Sockets API που διατίθεται για την γλώσσα C. Έτσι λοιπόν φτιάχνουμε για τον server και τον client προγραμματιστικές διεπαφές για επικοινωνία πάνω από TCP/IP, οι οποίες ονομάζονται sockets. Αφού τις φτιάξουμε, "δένουμε" το socket του server με μια IP και μια θύρα (port) και του δίνουμε την ιδιότητα να "ακούει" (listen) τις εισερχόμενες συνδέσεις, έτσι δηλαδή όπως δουλεύει κάθε server στον internet. Από την πλευρά του client, δίνουμε την εντολή να συνδεθεί (connect) με το socket (IP and port) του server.

Μόλις εγκατασταθεί η σύνδεση μεταξύ των δυο κόμβων, πρέπει να μπορούν να επικοινωνήσουν. Αυτό από πλευράς διεργασίας (αφού η εφαρμογή μας είναι μια διεργασία για το λειτουργικό) σημαίνει ότι θα πρέπει να στείλει ένα μήνυμα προς την άλλην πλευρά επικοινωνίας, ή να εμφανίσει ένα μήνυμα στην κονσόλα της στην περίπτωση που λήφθηκε ένα μήνυμα. Αυτό κατ'επέκταση σημαίνει ότι η διεργασία θα πρέπει ταυτόχρονα να ακούει από το standad input και από το socket, ώστε να ενεργήσει κάθε φορά με διαφορετικό τρόπο. Αυτό επιτυγχάνεται με την χρήση σύγχρονης I/O πολυεπεξεργασίας, και την συνάρτηση select. Αυτή ουσιαστικά κάνει read() από ένα σύνολο file decriptors (και το socket ως fd αναπαριστάται) και ανάλογα με το ποια πηγή έχει ενεργοποιήσει την read() δρα ανάλογα.

2.2 Ζητούμενο 2: Κρυπτογραφημένο chat πάνω από TCP/IP

Στο σημείο αυτό επεκτείνουμε το παραπάνω εργαλείο chat, ώστε τα δεδομένα που μεταφέρονται πάνω από TCP/IP να είναι κρυπτογραφημένα με προσυμφωνημένο κλειδί. Αυτό επιτυγχάνεται με την χρήση του cryptodev-linux API από userspace, που αναφέραμε και προηγουμένως. Αναγκαίο για την εφαρμογή είναι να είναι εγκαταστημένο το cryptodev module ώστε να μπορούμε να χρησιμοποιήσουμε το API και να έχουμε πρόσβαση στη συσκευή /dev/crypto. Όπως αναφέραμε και προηγουμένως για την κρυπτογράφηση των δεδομένων χρησιμοποιούμε κλήσεις ioctl στο /dev/crypto, ώστε ο crypto driver να κάνει χρήση του υλικού για κρυπτογράφηση. Πλέον ο κώδικας των δυο διεργασιών (client-server) παίρνει την τελική του μορφή.

Η υλοποίηση των ζητουμένων 1 και 2 για τον server και τον client ακολουθεί:

socket-client.c

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

#define BLOCK_SIZE 16
#define KEY_SIZE 16
#define DATA_SIZE 256
unsigned char buf[DATA_SIZE];
unsigned char key[]= "secretkeyabcdef";
unsigned char iv[] = "secretkeyabcdef";
struct session_op sess;

void toupper_buf(unsigned 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 DoEncryption(int cfd){
   struct crypt_op cryp;
   struct {
      unsigned char  in[DATA_SIZE],
           encrypted[DATA_SIZE],
```

```
            iv[BLOCK_SIZE];
    }data;
    memset(&cryp,0,sizeof(cryp));
    int i;

    //printf("%s", "This is the data before encrypt\n");
    //for (int i=0;i<DATA_SIZE;i++) printf("%c",buf[i]);
    //printf("\n\n");

    cryp.ses = sess.ses;
    cryp.len = sizeof(buf);
    cryp.src = buf;
    cryp.dst = data.encrypted;
    cryp.iv = iv;
    cryp.op = COP_ENCRYPT;
    if(ioctl(cfd, CIOCCRYPT, &cryp)){
        perror("ioctl encrypt");
        return 1;
    }

    //printf("%s","This is the encrypted data\n");
    //for (int i=0; i<DATA_SIZE; i++) printf("%x",data.encrypted[i]);
    //fprintf(stdout,"\n\n");

    i=0;
    memset(buf,'\0',DATA_SIZE);
    while (data.encrypted[i] != '\0'){
        buf[i] = data.encrypted[i];
        //printf("%x",buf[i]);
        i++;
    }
    //printf("\n\n");
    //printf("%x\n",data.encrypted[i]);


    //fprintf(stdout,"%s", "This is the buff after encrypt\n");
    //for (int i=0; i<DATA_SIZE; i++) printf("%c",buf[i]);

    return 0;
}

int DoDecryption(int cfd){
    struct crypt_op cryp;
    memset(&cryp,0,sizeof(cryp));
    struct {
        unsigned char  in[DATA_SIZE],
              decrypted[DATA_SIZE],
              iv[BLOCK_SIZE];
    }data;
    cryp.ses = sess.ses;
    cryp.len = sizeof(buf);
    cryp.src = buf;
    cryp.dst = data.decrypted;
```

```
    cryp.iv = iv;
    cryp.op = COP_DECRYPT;
    if (ioctl(cfd,CIOCCRYPT,&cryp)){
        perror("ioctl decrypt");
        return 1;
    }
    int i=0;
    memset(buf, '\0', DATA_SIZE);
    while (data.decrypted[i]!= '\0'){
        buf[i] = data.decrypted[i];
        i++;
    }
    return 0;
}


int main(int argc, char *argv[])
{
    int sd, port;
    ssize_t n;
    char *hostname;
    struct hostent *hp;
    struct sockaddr_in sa;
    fd_set set;
    FD_ZERO(&set); //initialize the set as empty

    memset(&sess,0,sizeof(sess));

    if (argc != 3) {
        fprintf(stderr, "Usage: %s hostname port\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");

/*
 * Start Crypto Sesson
 */
int cfd = open("/dev/crypto",O_RDWR);
if (cfd < 0){
   perror("open /dev/crypto");
   exit(1);
}

sess.cipher = CRYPTO_AES_CBC;
sess.keylen = KEY_SIZE;
sess.key = key;
if(ioctl(cfd, CIOCGSESSION, &sess)){
   perror("ioctl(session)");
   return 1;
}

/* Be careful with buffer overruns, ensure NUL-termination */

buf[0] = 'h';
buf[1] = 'e';
buf[2] = 'l';
buf[3] = 'l';
buf[4] = 'o';
buf[5] = '\n';
buf[6] = '\0';
fprintf(stdout, "I said:\n%sRemote says:\n",buf);
fflush(stdout);
/* Say something... */
DoEncryption(cfd);
if (insist_write(sd, buf, sizeof(buf)) != sizeof(buf)) {
   perror("write");
   exit(1);
}

/* Let the remote know we're not going to write anything else.
 * Try removing the shutdown() call and see what happens.
 */
//if (shutdown(sd, SHUT_WR) < 0) {
// perror("shutdown");
// exit(1);
//}

/* Read answer and write it to standard output */
for (;;) {
   memset(&buf,'\0',sizeof(buf));
   FD_SET(0,&set); //add to read set the stin
```

```
      FD_SET(sd,&set); //add to read set the socket
      if (select(sd+1,&set,NULL,NULL,NULL)<0) perror("pselect");
      if (FD_ISSET(sd,&set)){
         n = read(sd, buf, sizeof(buf));
         if (DoDecryption(cfd)) perror("decryption");
         if (n < 0) {
            perror("read");
            exit(1);
         }

         if (n <= 0)
            break;
         toupper_buf(buf,sizeof(buf));
         if (insist_write(1, buf, sizeof(buf)) != sizeof(buf)) {
            perror("write");
            exit(1);
         }
      continue;
      }
      n = read(0,buf,sizeof(buf));
      if (DoEncryption(cfd)) perror("encryption");
      if (n < 0) {
         perror("error");
         exit(2);
      }
      if (n<=0) break;
      if (insist_write(sd,buf,sizeof(buf)) != sizeof(buf) ){
         perror("write");
         exit(1);
      }
   }
   fprintf(stderr, "\nDone.\n");
   if (ioctl(cfd,CIOCFSESSION,&sess.ses)){
      perror("ioctl fsession");
      return 1;
   }
   if (close(cfd)<0){
      perror("close fd");
      return 1;
   }
   return 0;
}
```

socket-server.c

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

#include <stdio.h>
```

```c
#include <errno.h>
#include <ctype.h>
#include <string.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>
#include <netdb.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <sys/stat.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 "crypto/cryptodev.h"
#define BLOCK_SIZE 16
#define KEY_SIZE 16
#define DATA_SIZE 256

unsigned char buf[DATA_SIZE];
unsigned char key[] = "secretkeyabcdef";
unsigned char iv[] = "secretkeyabcdef";
struct session_op sess;

/* Convert a buffer to upercase */
void toupper_buf(unsigned 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 DoEncryption(int cfd){
   struct crypt_op cryp;
   memset(&cryp,0,sizeof(cryp));
   struct{
      unsigned char  in[DATA_SIZE],
           encrypted[DATA_SIZE],
           iv[BLOCK_SIZE];
   }data;

   cryp.ses = sess.ses;
   cryp.len = sizeof(buf);
   cryp.src = buf;
   cryp.dst = data.encrypted;
   cryp.iv = iv;
   cryp.op = COP_ENCRYPT;
   if(ioctl(cfd,CIOCCRYPT,&cryp)){
      perror("ioctl crypt");
      return 1;
   }
   int i=0;
   memset(buf,'\0',DATA_SIZE);
   while(data.encrypted[i] != '\0'){
      buf[i] = data.encrypted[i];
      i++;
   }
   return 0;
}
int DoDecryption(int cfd){
   struct crypt_op cryp;
   struct{
      unsigned char  in[DATA_SIZE],
           decrypted[DATA_SIZE],
           iv[BLOCK_SIZE];
   }data;


   memset(&cryp,0,sizeof(cryp));
   cryp.ses = sess.ses;
   cryp.len = sizeof(buf);
   cryp.src = buf;
   cryp.dst = data.decrypted;
   cryp.iv = iv;
   cryp.op = COP_DECRYPT;
   if (ioctl(cfd,CIOCCRYPT,&cryp)){
      perror("ioctl decrypt");
      return 1;
   }
   memset(buf,'\0', DATA_SIZE);
   int i=0;
   while (data.decrypted[i]!= '\0'){
      buf[i] = data.decrypted[i];
      i++;
```

```
    }
    return 0;
}
int main(void)
{

    char addrstr[INET_ADDRSTRLEN];
    int sd, newsd;
    ssize_t n;
    socklen_t len;
    struct sockaddr_in sa;
    fd_set set;
    FD_ZERO(&set);
    memset(&sess,0,sizeof(sess));
    memset(&buf,'\0',sizeof(buf));
    /* 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);
    }
    int cfd = open("/dev/crypto", O_RDWR);
    if (cfd < 0 ){
        perror("open /dev/crypto");
        exit(1);
    }
    sess.cipher = CRYPTO_AES_CBC;
    sess.keylen = KEY_SIZE;
    sess.key = key;
    if (ioctl(cfd,CIOCGSESSION,&sess)){
        perror("ioctl session");
        return 1;
    };
```

```
/* Loop forever, accept()ing connections */
for (;;) {
   memset(&buf,'\0',sizeof(buf));
   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 (;;) {
      memset(&buf,'\0',sizeof(buf));
      FD_SET(0,&set);
      FD_SET(newsd,&set);
      if(select(newsd+1,&set,NULL,NULL,NULL)<0) perror("pselect");
      if (FD_ISSET(newsd,&set)){
         n = read(newsd,buf,sizeof(buf));
         if (DoDecryption(cfd)) perror("decryption");
         if (n < 0){
            perror("read");
            exit(1);
         }
         if (n <= 0) break;
         toupper_buf(buf, sizeof(buf));
         if (insist_write(1,buf,sizeof(buf)) != sizeof(buf)){
            perror("write");
            break;
         }
      continue;

      }
      n = read(0,buf,sizeof(buf));
      if (DoEncryption(cfd)) perror("encryption");
      if (n <= 0) {
         if (n < 0)
            perror("read from remote peer failed");
         else
            fprintf(stderr, "Peer went away\n");
         break;
      }
      if (insist_write(newsd, buf, sizeof(buf)) != sizeof(buf)) {
         perror("write to remote peer failed");
         break;
```

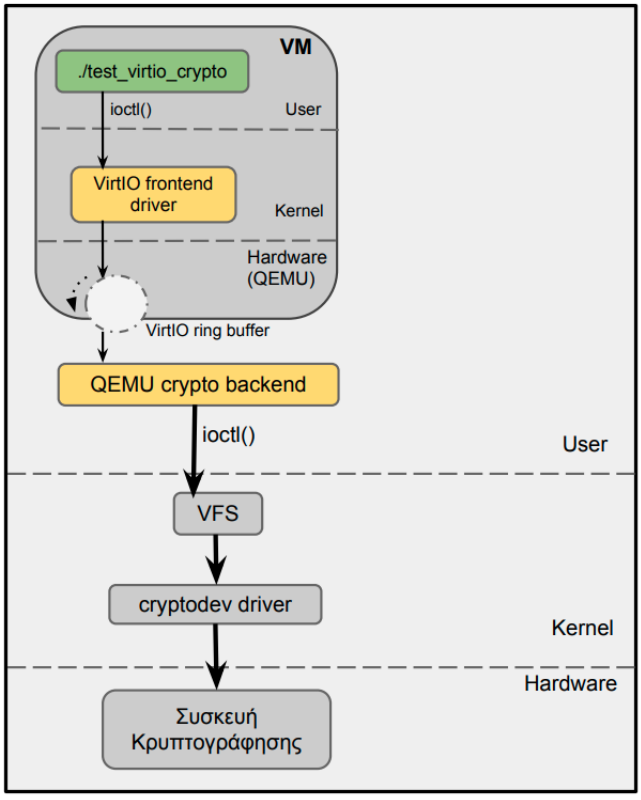
```
        }
      }

      /* Make sure we don't leak open files */
      if (close(newsd) < 0)
        perror("close");
    }
    if (close(cfd)<0) perror("close cfd");
    if (ioctl(cfd,CIOCFSESSION, &sess.ses)){
      perror("ioctl fsession");
      return 1;
    }
    /* This will never happen */
    return 1;
}
```

Η διαδικασία της κρυπτογράφησης και της αποκρυπτογράφησης γίνονται στις συναρτήσεις DoEncryptio και DoDecryption, στις οποίες βλέπουμε και τον τρόπο που γίνεται η χρήση του cryptodev-linux API και των ioctl κλήσεων στο ειδικό αρχείο /dev/crypto.

2.3 Ζητούμενο 3: Υλοποίηση συσκευής cryptodev με VirtIO

Στο σημείο αυτό σχεδιάζουμε και υλοποιούμε την εικονική συσκευή cryptodev, με χρήση προτύπου VirtIO, έτσι ώστε το προηγούμενο εργαλείο να μπορεί να εκτελείται μέσα σε VM κάνοντας χρήση κρυπτογραφικών επιταχυντών σε υλικό, οι οποίοι ως τώρα ήταν προσβάσιμοι μόνο από τον host. Στην ουσία χρησιμοποιούμε την τεχνική της παρα-εικονοποίησης (paravirtualization). Για την επίτευξη του σκοπού μας, είναι απαραίτητο να τροποποιηθεί και κομμάτι του πυρήνα του guest (VM) και να εισαχθεί driver ο οποίος θα αναλαμβάνει την επικοινωνία με τον hypervisor (QEMU-KVM) μέσω hypercalls χρησιμοποιώντας στην εικονική συσκευή cryptodev.

Το πρότυπο VirtIO επιτρέπει την αποδοτική επικοινωνία του ΛΣ που εκτελείται μέσα στην εικονική μηχανή με κώδικα που εκτελείται στον host, μέσω του ορισμού κοινών πόρων για ανταλλαγή δεδομένων.Αυτοί οι κοινοί πόροι ονομάζονται VirtQueues. Στο πρώτο μέρος η κρυπτογράφηση/αποκρυπτογράφηση γινόταν με κατάλληλες κλήσεις ioctl() στην συσκευή /dev/crypto. Τώρα θα πρέπει οι κλήσεις ioctl() να μεταφέρονται από τον guest στον hypervisor μέσω των virtqueues, ο hypervisor καλεί την αντίστοιχη ioctl() για την πραγματική συσκευή και επιστρέφει το αποτέλεσμα στον guest πάλι μέσω των virtqueues. Αυτή η αλληλουχία κλήσεων φαίνεται και στο σχήμα:



Σχήμα 2: Αρχιτεκτονική λογισμικού της (paravirtualized) virtio-cryptodev συσκευής

Από το σχήμα γίνεται προφανής ο διαχωρισμός του οδηγού σε δύο μέρη, backend και frontend. Το πρώτο μέρος εκτελείται στο χώρο χρήστη του host και είναι μέρος του QEMU-KVM, ενώ το δεύτερο στο χώρο πυρήνα της εικονικής μηχανής ως driver του VM. Τα δυο μέρη επικοινωνούν μετακύ τους μέσω VirtQueues.

2.3.1 Userspace Guest

Αρχικά ξεκινώντας την παρουσίαση από πάνω προς τα κάτω σύμφωνα με το σχήμα 2, παρουσιάζεται ο κώδικας που τρέχει ως διεργασία στο userspace του VM.

test_crypto.c

```
/*
 * test_crypto.c
 *
 * Performs a simple encryption-decryption of urandom data from /dev/urandom
 * with the use of cryptodev device.
 *
 * Stefanos Gerangelos <sgerag@cslab.ece.ntua.gr>
 *
 */

#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>

#include <stdlib.h>
#include <string.h>
#include <sys/ioctl.h>
#include "cryptodev.h"

#include <sys/types.h>
#include <sys/stat.h>

#define DATA_SIZE    16384
#define BLOCK_SIZE   16
#define KEY_SIZE     24


unsigned char key[] = "abcdefghijklmnopqrstuvw";
int fill_urandom_buff(unsigned char in[DATA_SIZE]);

static int test_crypto(int cfd)
{
   struct session_op sess;
   struct crypt_op cryp;
   struct {
      unsigned char in[DATA_SIZE],
           encrypted[DATA_SIZE],
           decrypted[DATA_SIZE],
           iv[BLOCK_SIZE],
           key[KEY_SIZE];
   } data;

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

   if (fill_urandom_buff(data.in) < 0) {
      printf("error @filling urandom data\n");
```

```
    return 1;
}

//printf("Sleeping for 30 seconds. Its time to remove the host device.\n");
//sleep(30);
//printf("Woke up after 30 seconds.\n");

/**
 * Get crypto session for AES128
 **/
sess.cipher = CRYPTO_AES_CBC;
sess.keylen = KEY_SIZE;
sess.key = (__u8 __user *)key;

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

/**
 * Encrypt data.in to data.encrypted
 **/
printf("Doing encryption of %d bytes of data...", DATA_SIZE);
fflush(stdout);
cryp.ses = sess.ses;
cryp.len = sizeof(data.in);
cryp.src = (__u8 __user *)data.in;
cryp.dst = (__u8 __user *)data.encrypted;
cryp.iv = (__u8 __user *)data.iv;
cryp.op = COP_ENCRYPT;
if (ioctl(cfd, CIOCCRYPT, &cryp)) {
    perror("ioctl(CIOCCRYPT)");
    return 1;
}
printf("[OK]\n");
/**
 * Decrypt data.encrypted to data.decrypted
 **/
printf("Doing decryption of %d bytes of data...", DATA_SIZE);
fflush(stdout);

cryp.src = (__u8 __user *)data.encrypted;
cryp.dst = (__u8 __user *)data.decrypted;
cryp.op = COP_DECRYPT;
if (ioctl(cfd, CIOCCRYPT, &cryp)) {
    perror("ioctl(CIOCCRYPT)");
    return 1;
}
printf("[OK]\n");

/**
 * Verify the result
 **/
```

```
    printf("Doing Verification of data...");
    fflush(stdout);
    if (memcmp(data.in, data.decrypted, sizeof(data.in)) != 0) {
        printf(" Error\n");
        return 1;
    } else {
        printf(" Success\n");
    }

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

    return 0;
}

int fill_urandom_buff(unsigned char in[DATA_SIZE]){
    int crypto_fd = open("/dev/urandom", O_RDONLY);
    int ret = -1;

    if (crypto_fd < 0)
        return crypto_fd;

    ret = read(crypto_fd, (void *)in, DATA_SIZE);

    close(crypto_fd);

    return ret;
}

int main(int argc, char **argv)
{
    int fd = -1;
    char *filename;
    char error_str[100];

    filename = (argv[1] == NULL) ? "/dev/crypto" : argv[1];
    fd = open(filename, O_RDWR, 0);
    printf("DONE\n");
    if (fd < 0) {
        sprintf(error_str, "open %s", filename);
        perror(error_str);
        return 1;
    }

    if (test_crypto(fd))
        return 1;

    if (close(fd)) {
```

```
        perror("close(fd)");
        return 1;
    }

    return 0;
}
```

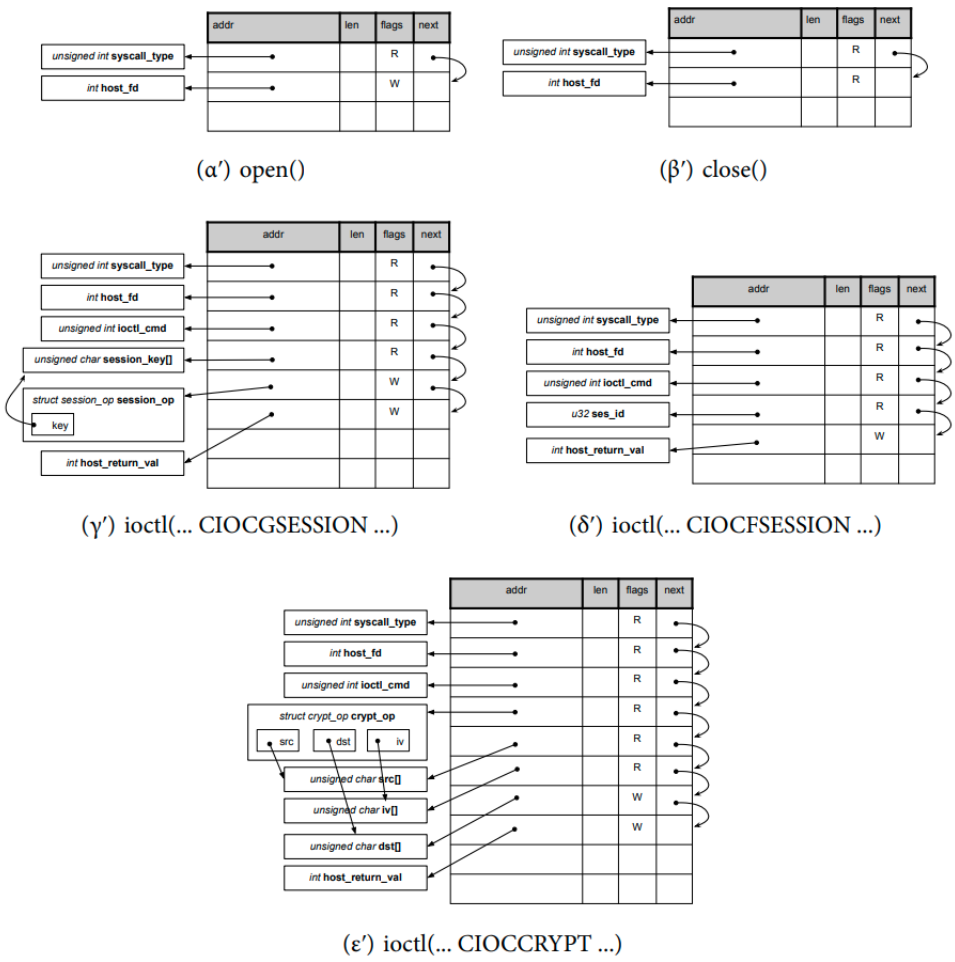
Βλέπουμε ότι η συγκεκριμένη διεργασία τρέχει κανονικά σαν να μην γνωρίζει τίποτα περί virtualization στο userspace του guest. Για την κρυπτογράφηση και αποκρυπτογράφηση χρησιμοποιείται η εικονική συσκευή /dev/cryptodev0 και κάνοντας χρήση των γνωστών ioctl κλήσεων σύμφωνα με το cryptodev-linux API.

2.3.2 Kernelspace Guest (Frontend)

Ως γνωστόν, μετά την κλήση ioctl στη εικονική κρυπτογραφική συσκευή cryptodev0, τον έλεγχο παίρνει ο character device driver της συσκευής, ο οποίος υλοποιεί τις κλήσεις αυτές. Όπως αναφέραμε προηγουμένως σκοπός του driver του guest είναι να κάνει hypercall στον hypervisor (QEMU-KVM). Η επικοινωνία αυτή επιτυγχάνεται μέσω του πρωτοκόλλου επικοινωνίας virtio-ring. Συνοπτικά, το frontend στην περίπτωσή μας γεμίζει λίστες scatter-gather και τις ”σπρώχνει” στην virtqueue, ενημερώνοντας το backend για τη συγκεκριμένη κίνηση. Στην συνέχεια περιμένει την απάντηση από το backend με τα επεξεργασμένα δεδομένα, και αφού τα λάβει, τα στέλνει στο userpsace του VM.

Πολύ ιδιαίτερη προσοχή δόθηκε στον driver του guest, κατά την ασφαλή αντιγραφή των δεδομένων που έρχονται από το userspace. Συγκεκριμένα, σε κάθε διαφορετικό command ioctl (CIOCCRYPT, CIOCGSESSION, CIOCFSESSION), οι δομές session_op και crypt_op περιέχουν pointers που αναφέρονται σε εικονική μνήμη της διεργασίες. Έτσι, πρέπει πέρα από την ασφαλή αντιγραφή των δομών αυτών, πρέπει να γίνει και ασφαλή αντιγραφή των δεδομένων στα οποία δείχνουν οι pointers του αρχικού struct. Η εντολή αυτή γίνεται μέσω του copy_from_user(). Επίσης, όταν γίνεται επεξεργασία της virtqueue ουράς από τον host, πρέπει να απομονώνουμε τον driver, ώστε να μην έχει πρόσβαση άλλη διεργασία στην ουρά και αλλάξει τα δεδομένα μας (aka rational conditions). Για το σκοπό αυτό κάθε φορά που κάνουμε το hypercall χρησιμοποιούμε έναν spinlock.

Η σύμβαση που ακολουθούμε για την δημιουργία των scatter-gather lists που στέλνονται στο backend είναι η ακόλουθη:



(α′) open()

(β′) close()

(γ′) ioctl(... CIOCGSESSION ...)

(δ′) ioctl(... CIOCFSESSION ...)

(ε′) ioctl(... CIOCCRYPT ...)

Σχήμα 3: Δομή Virtqueue που χρησιμοποιεί η συσκευή virtio-cryptodev-pci για κάθε κλήση συστήματος

Η υλοποίηση του frontend για το kernelspace του guest είναι η ακόλουθη:

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>
#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_out = 0,num_in = 0;
   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];

   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;

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.
 **/
/* ?? */
sg_init_one(&syscall_type_sg,syscall_type,sizeof(*syscall_type));
sgs[num_out++] = &syscall_type_sg;

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

virtqueue_add_sgs(crdev->vq, sgs, num_out, num_in, &syscall_type_sg, GFP_ATOMIC);
virtqueue_kick(crdev->vq);
/**
 * Wait for the host to process our data.
 **/
/* ?? */

while(virtqueue_get_buf(crdev->vq,&len) == NULL);
/* If host failed to open() return -ENODEV. */
/* ?? */
if ((crof->host_fd=*host_fd) <= 0){
   ret = -ENODEV;
   goto fail;
}
```

```
fail:
   debug("Leaving");
   return ret;
}

static int crypto_chrdev_release(struct inode *inode, struct file *filp)
{
   int ret = 0;
   unsigned int num_out = 0,num_in = 0;
   struct crypto_open_file *crof = filp->private_data;
   struct crypto_device *crdev = crof->crdev;
   unsigned int *syscall_type,len;
   struct scatterlist syscall_type_sg,host_fd_sg, *sgs[2];
   debug("Entering");

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

   /**
    * Send data to the host.
    **/
   /* ?? */
   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;

   virtqueue_add_sgs(crdev->vq, sgs, num_out, num_in, &syscall_type_sg, GFP_ATOMIC);
   virtqueue_kick(crdev->vq);
   /**
    * Wait for the host to process our data.
    **/
   /* ?? */
   while(virtqueue_get_buf(crdev->vq,&len) == NULL);
   kfree(crof);
   debug("Leaving");
   return ret;

}

static long crypto_chrdev_ioctl(struct file *filp, unsigned int cmd,
                                unsigned long arg)
{
   long ret = 0;
   uint32_t *id;
   int err;
        int *host_return_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, host_fd_sg, ioctl_cmd_sg, session_key_sg,
       session_op_sg, host_return_cal_sg, output_msg_sg, input_msg_sg, crypt_op_sg,
       src_sg, iv_sg, dst_sg, host_return_val_sg, ses_id_sg,
                    *sgs[12];
   unsigned int num_out, num_in, len;
   int i;
#define MSG_LEN 100
   unsigned char *output_msg, *input_msg, *session_key, *src, *iv, *dst;
   unsigned int *syscall_type,*command;
   struct session_op *session_op,*sess_ptr;
   struct crypt_op *crypt_op,*crypt_ptr;
   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;

   command = kzalloc(sizeof(cmd),GFP_KERNEL);
   *command = cmd;

   crypt_op = kzalloc(sizeof(struct crypt_op), GFP_KERNEL);
   session_op = (struct session_op *)kzalloc(sizeof(struct session_op), GFP_KERNEL);
   host_return_val = kzalloc(sizeof(int),GFP_KERNEL);

   id = kzalloc(sizeof(uint32_t),GFP_KERNEL);

   num_out = 0;
   num_in = 0;

   dst = NULL;
   sess_ptr = NULL;
   crypt_ptr = 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;

   /**
    *  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(&ioctl_cmd_sg, command, sizeof(*command));
    sgs[num_out++] = &ioctl_cmd_sg;
    sess_ptr = (struct session_op *)arg;
    debug("1");
    if (copy_from_user(session_op, sess_ptr, sizeof(struct session_op))){
      debug("Copy from user");
      ret = -1;
      goto fail;
    }
    debug("2");
    session_key = kzalloc(session_op->keylen*sizeof(unsigned char), GFP_KERNEL);
    if (!session_key) {
      ret = -ENOMEM;
      goto fail;
    }
    if (copy_from_user(session_key, session_op->key, sizeof(*session_key))){
      debug("Copy from user");
      ret = -1;
      goto fail;
    }
    debug("3");
    session_op->key = session_key;
    sg_init_one(&session_key_sg, session_op->key, sizeof(unsigned
        char)*session_op->keylen);
    sgs[num_out++] = &session_key_sg;

    sg_init_one(&output_msg_sg, output_msg, MSG_LEN);
    sgs[num_out++] = &output_msg_sg;

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

    sg_init_one(&session_op_sg, session_op, sizeof(struct session_op));
    sgs[num_out + num_in++] = &session_op_sg;

    sg_init_one(&host_return_val_sg, host_return_val, sizeof(*host_return_val));
    sgs[num_out + num_in++] = &host_return_val_sg;
    debug("4");
    break;

  case CIOCFSESSION:
    debug("CIOCFSESSION");
    memcpy(output_msg, "Hello HOST from ioctl CIOCFSESSION.", 36);
    input_msg[0] = '\0';

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

    if (copy_from_user(id, (uint32_t*)arg, sizeof(*id))){
```

```
        debug("Copy from user");
        ret = -1;
        goto fail;
    }
    sg_init_one(&ses_id_sg, id, sizeof(*id));
    sgs[num_out++] = &ses_id_sg;

    sg_init_one(&output_msg_sg, output_msg, MSG_LEN);
    sgs[num_out++] = &output_msg_sg;

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

    sg_init_one(&host_return_val_sg, host_return_val, sizeof(*host_return_val));
    sgs[num_out + num_in++] = &host_return_val_sg;

    break;

case CIOCCRYPT:
    debug("CIOCCRYPT");
    memcpy(output_msg, "Hello HOST from ioctl CIOCCRYPT.", 33);
    input_msg[0] = '\0';

    sg_init_one(&ioctl_cmd_sg, command, sizeof(*command));
    sgs[num_out++] = &ioctl_cmd_sg;
    debug("1");
    crypt_ptr = (struct crypt_op *)arg;
    if (copy_from_user(crypt_op, crypt_ptr, sizeof(struct crypt_op))){
        debug("Copy from user 1");
        ret = -1;
        goto fail;
    }



    debug("2");
    src = kzalloc(crypt_op->len*sizeof(unsigned char),GFP_KERNEL);
    if (!src){
        ret = -ENOMEM;
        goto fail;
    }
    if (copy_from_user(src, crypt_op->src, sizeof(unsigned char)*crypt_op->len)){
        debug("Copy from user 2");
        ret = -1;
        goto fail;
    }


    debug("3");

    iv = kzalloc(16*sizeof(unsigned char),GFP_KERNEL);
    if (!iv){
```

```
        ret = -ENOMEM;
        goto fail;
    }
    if (copy_from_user(iv, crypt_op->iv,sizeof(16*sizeof(unsigned char)))){
        debug("Copy from user 3");
        ret = -1;
        goto fail;
    }
    dst = kzalloc(crypt_op->len*sizeof(unsigned char), GFP_KERNEL);
    if (!dst){
        ret = -ENOMEM;
        goto fail;
    }
    if (copy_from_user(dst,crypt_op->dst, sizeof(crypt_op->len*sizeof(unsigned
         char)))){
        debug("Copy from user dst");
        ret = -1;
        goto fail;
    }
    debug("4");
    /*crypt_op->iv = iv;
    crypt_op->dst = dst;
          crypt_op->src = src;*/
    printk("Crypto src is: ");
    for (i=0 ; i <crypt_op->len;i++ ) printk("%x",crypt_op->src[i]);
    printk("\n");
    sg_init_one(&crypt_op_sg, crypt_op, sizeof(struct crypt_op));
    sgs[num_out++]= &crypt_op_sg;

    sg_init_one(&src_sg,src, crypt_op->len *sizeof(unsigned char));
    sgs[num_out++]=&src_sg;

    sg_init_one(&iv_sg, iv, sizeof(unsigned char)*16);
    sgs[num_out++] = &iv_sg;

    sg_init_one(&output_msg_sg, output_msg, MSG_LEN);
    sgs[num_out++] = &output_msg_sg;

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

    sg_init_one(&dst_sg,dst,sizeof(unsigned char)*crypt_op->len);
    sgs[num_out + num_in++] = &dst_sg;
    debug("5");

    sg_init_one(&host_return_val_sg, host_return_val, sizeof(*host_return_val));
    sgs[num_out + num_in++] = &host_return_val_sg;

    break;

default:
    debug("Unsupported ioctl command");
```

```
   break;
}


/**
 * Wait for the host to process our data.
 **/
/* ?? */
/* ?? Lock ?? */
spin_lock_irqsave(&crdev->lock, flags);
debug("5");
err = virtqueue_add_sgs(vq, sgs, num_out, num_in,
                     &syscall_type_sg, GFP_ATOMIC);
if (err) {
   debug("error");
   ret = -1;
   goto fail;
}
debug("6");
virtqueue_kick(vq);
debug("7");
while (virtqueue_get_buf(vq, &len) == NULL);
   /* do nothing */
spin_unlock_irqrestore(&crdev->lock, flags);

debug("8");
switch(cmd){
case CIOCGSESSION:
   debug("CIOCGSESSION RETURN");
   if (copy_to_user(sess_ptr,session_op,sizeof(struct session_op))){
      debug("Copy to user");
      ret = -1;
      goto fail;
   }
   break;

case CIOCFSESSION:
   debug("CIOCFSESSION RETURN");
   break;

case CIOCCRYPT:
   debug("CIOCCRYPT RETURN");

   if (copy_to_user( crypt_ptr->dst , dst, crypt_op->len*sizeof(unsigned char))){
      debug("Copy to User 10");
      ret = -1;
      goto fail;
   }
   break;

}
```

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

    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");
}
```

2.3.3 Userspace Host (Backend)

Την σειρά αναλαμβάνει η διεργασία QEMU που τρέχει στο userspace του Host. Αρχικά, όποιες κλήσεις λαμβάνονται από την εικονική συσκευή cryptodev του VM, διαχειρίζονται από κώδικα του QEMU ο οποίος είναι και ο hypervisor του VM. Έτσι λοιπόν όταν λαμβάνει δράση η διεργασία QEMU, σημαίνει ότι το frontend έχει γεμίσει την virtqueue. Τότε, το QEMU παίρνει τα στοιχεία της ουράς. Ανάλογα με την κλήση ioctl και την σύμβαση της λίστας που ακολουθούμε, διαβάζει τα δεδομένα. Σκοπός της είναι να καλέσει την κρυπτογραφική συσκευή /dev/crypto του host μηχανήματος, ώστε να χρησιμοποιηθούν οι επιταχυντές υλικού του host. Για τον λόγο αυτό πρέπει να επανασυναρμολογήσει τα ληφθέντα δεδομένα και να φτιάξει δομές κατάλληλες και έγκυρες για την κλήση ioctl προς τη κρυπτογραφική συσκευή. Αφού λάβει λοιπόν τα αποτελέσματα της κλήσης, πρέπει να στείλει τα δεδομένα προς την εικονική κρυπτογραφική συσκευή του frontend.

Η υλοποίηση του backend για το userspace του host είναι η ακόλουθη:

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>
#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;
    int *host_fd,*host_return_val;
    struct crypt_op *crypt_op;
    int i;
    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);
if (*host_fd < 0) {
   perror("open(dev/crypto)");
   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)){
   perror("close");
   return ;
}
     /* ?? */
     break;

 case VIRTIO_CRYPTODEV_SYSCALL_TYPE_IOCTL:
     DEBUG("VIRTIO_CRYPTODEV_SYSCALL_TYPE_IOCTL");
     /* ?? */
     //unsigned char *output_msg = ;
     unsigned char *input_msg = elem->in_sg[0].iov_base;
     memcpy(input_msg, "Host: Welcome to the virtio World!", 35);
     //printf("Guest says: %s\n", output_msg);
     printf("We say: %s\n", input_msg);
ioctl_cmd = elem->out_sg[2].iov_base;
host_fd = elem->out_sg[1].iov_base;
switch (*ioctl_cmd){
case CIOCGSESSION:
   DEBUG("CIOCGSESSION BACKEND");
   struct session_op *session_op = (struct session_op *) elem->in_sg[1].iov_base;
   host_return_val = elem->in_sg[2].iov_base;
   session_op->key = elem->out_sg[3].iov_base;
   if (ioctl(*host_fd,CIOCGSESSION,session_op)){
      *host_return_val = - 1;
      perror("error back ioctl(CIOCGSESSION)");
   }
   else *host_return_val = 0;
   //printf("The session key is --> %s\n",session_op->key);
   break;
case CIOCFSESSION:
   DEBUG("CIOCFSESSION");
   uint32_t * ses_id;
   ses_id = elem->out_sg[3].iov_base;
   host_return_val = elem->in_sg[1].iov_base;
   if (ioctl(*host_fd, CIOCFSESSION, ses_id)){
      *host_return_val = -1;
      perror("Back ioctl(CIOCFSESSION)");
```

```
      }
      else *host_return_val = 0;
      break;
    case CIOCCRYPT:
      DEBUG("CIOCCRYPT");
      host_return_val = elem->in_sg[2].iov_base;

      crypt_op = (struct crypt_op *)elem->out_sg[3].iov_base;

      crypt_op -> dst = elem->in_sg[1].iov_base;
      crypt_op -> src = elem->out_sg[4].iov_base;
      crypt_op -> iv = elem->out_sg[5].iov_base;
      /*printf("Input Data: ");
      for (i=0; i<crypt_op->len; i++)printf("%x",crypt_op->src);*/
      *host_return_val = ioctl(*host_fd,CIOCCRYPT,crypt_op);
      if (*host_return_val){
         perror("error Back ioctl(CIOCCRYPT)");
      }
      /*printf("Ouput Data: ");
      for (i=0; i<crypt_op->len; i++) printf("%x",crypt_op->dst);
      printf("\n");*/
      break;
    default:
      DEBUG("Uknown ioctl cmd");
      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)
```

Έτσι λοιπόν έχουμε πετύχει την επικοινωνία του userspace ενός VM με το υλικό του Host με χρήση της τεχνικής paravirtualization. Επομένως μπορούμε να συνδέσουμε όλα τα παραπάνω και να λάβουμε μια εφαρμογή chat που χρησιμοποιεί επικοινωνία πάνω από TCP/IP με κρυπτογραφημένα μηνύματα μεταξύ δύο VMs, τα οποία χρησιμοποιούν τους επιταχυντές κρυπτογράφησης του υλικού των host μηχανημάτων τους.