Αναφορά της τρίτης εργαστηριακής άσκησης

Ανθοπούλου Φαίδρα-Αναστασία 03118818
Πευκιανάκης Κωνσταντίνος 03114897

Οι προσθήκες που κάναμε, βρίσκονται στα αρχεία socket-client.c, socket-server.c, socket-common.h, η οποίες αφορούν στα ζητούμενα Z1 και Z2 της εργασίας, καθώς και στα crypto-chrdev.c, crypto.h, crypto-module.c, virtio-cryptodev.c η οποίες αφορούν στο ζητούμενο Z3 της εργασίας. Για το ζητούμενο Z3 τα αρχεία crypto-chrdev.c, crypto.h, crypto-module.c αναφέρονται στο front-end κομμάτι της συσκευής και το αρχείο virtio-cryptodev.c αναφέρεται στο back-end κομμάτι. Ακολουθούν οι συναρτήσεις που υλοποιήθηκαν στο κάθε αρχείο, και από κάτω μια περιγραφή σχετικά με την κάθε μία, καθώς και τυχόντα προβλήματα που αντιμετωπίσαμε.

Ζητούμενα Z1 και Z2

Στο socket-client.c:

Εδώ εκτός από την main(), υλοποιήσαμε τις cryptodev_init(), crypto_key(), encrypt(), και decrypt().

cryptodev_init():

```
int cryptodev_init(){

	cfd = open("/dev/crypto", O_RDWR);

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

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

	/*
	 * Get crypto session for AES128
	 */

	sess.cipher = CRYPTO_AES_CBC;
	sess.keylen = KEY_SIZE;
	sess.key = key;
	printf("Key set to: %s\n", key);

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

	for(int i = 0; i < BLOCK_SIZE; i++)
	iv[i] = 0x00;

	cryp.ses = sess.ses;
	cryp.iv = iv;

	printf("Cryptodev initialized! \n");
	return 0;
}
```

Σε αυτήν την συνάρτηση, ανοίγουμε αρχικά τον driver με δικαιώματα εγγραφής και ανάγνωσης και αρχικοποιούμε τα struct sess και crypt. Ορίζουμε ότι ο αλγόριθμος με τον οποίο θα γίνει η κρυπτογράφιση (cipher) είναι ο CRYPTO_AES_CBC, ότι το μήκος του κλειδιού που θα χρησιμοποιηθεί είναι KEY_SIZE (ορίζεται στην αρχή του προγράμματος να είναι 16 με την εντολή #define KEY_SIZE 16), και ότι το κλειδί θα είναι το key(ορίζεται στην συνάρτηση crypto_key που ακολουθεί). Μετά καλούμε την συνάρτηση ioctl για να εκτελέσει ο οδηγός την λειτουργία CIOCGSESSION (αρχή συνόδου κρυπτογράφισης). Τέλος, ορίζουμε ότι το session στο οποίο αναφέρεται το struct cryp είναι το sess, και το διάνυσμα αρχικοποίησης του (iv) είναι 16 μηδενικά (το BLOCK_SIZE ορίζεται κι αυτό στην αρχή με #define BLOCK_SIZE 16). Ο λόγος που ορίσαμε το KEY_SIZE ίσο με 16 είναι ότι το απαιτεί ο CRYPTO_AES_CBC, ενώ το BLOCK_SIZE ήταν ένα αυθαίρετο πολλαπλάσιο του 4.

crypto_key():

```
void crypto_key(char * str, int len){

	int i;
	for(i = 0; i < len; i++)
	key[i] = str[i];
	for(; i < KEY_SIZE; i++)
	key[i] = 0x00;
}
```

Αυτή είναι η συνάρτηση που ορίζει το κλειδί που θα χρησιμοποιήσουμε στην κρυπτογράφηση. Αρχικά παίρνουμε ένα string ως όρισμα και γεμίζουμε τους πρώτους len χαρακτήρες του key με αυτό. Αν το string και το len δεν είναι αρκετά μεγάλα για να γεμίσει όλο το key (που πρέπει να έχει μέγεθος 16), τότε γεμίζουμε τις εναπομένουσες θέσεις του με μηδενικά. Στο τέλος, το key έχει οριστεί και είναι έτοιμο για χρήση.

encrypt():

```
int encrypt(unsigned char * message, unsigned char * encrypted_message, int len){

	cryp.ses = sess.ses;
	cryp.iv = iv;
	cryp.len = len;
	cryp.src = message;
	cryp.dst = encrypted_message;
	cryp.op = COP_ENCRYPT;

	if (ioctl(cfd, CIOCCRYPT, &cryp)) {
		perror("ioctl(CIOCCRYPT)");
		return 1;
	}
	return 0;
}
```

Αυτή η συνάρτηση είναι αυτή που θα καλέσουμε για να κρυπτογραφήσουμε τα δεδομένα μας. Αρχικά δίνουμε τις κατάλληλες τιμές στα πεδία του cryp. Ορίζουμε πως το session που αναφέρεται το cryp είναι το sess, το iv θα είναι το iv που έχουμε ορίσει ήδη στην cryptodev_init(), το μήκος len θα είναι αυτό που δίνεται ως τρίτο όρισμα στην συνάρτηση, το μήνυμα που θα κρυπτογραφηθεί θα είναι το πρώτο όρισμα, ενώ το κρυπτογραφημένο

μήνυμα θα είναι το δεύτερο όρισμα. Επίσης ορίζουμε πως η λειτουργία που εκτελείται σε αυτήν την συνάρτηση είναι η COP_ENCRYPT. Τέλος, καλούμε τον οδηγό να κάνει την κρυπτογράφηση μέσω της ioctl.

decrypt():

```
int decrypt(unsigned char * encrypted_message, unsigned char * decrypted_message,
int len){

	cryp.len = len;
	cryp.src = encrypted_message;
	cryp.dst = decrypted_message;
	cryp.op = COP_DECRYPT;

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

	return 0;
}
```

Αυτή η συνάρτηση είναι αυτή που θα καλέσουμε για να αποκρυπτογραφήσουμε τα δεδομένα μας. Πάλι εδώ δίνουμε τις κατάλληλες τιμές στα πεδία του cryp. Παρομοιώς με την encrypt() ορίζουμε ότι το μήκος len θα είναι αυτό που δίνεται ως τρίτο όρισμα στην συνάρτηση, το μήνυμα που θα αποκρυπτογραφηθεί θα είναι το πρώτο όρισμα, ενώ το αποκρυπτογραφημένο μήνυμα θα είναι το δεύτερο όρισμα. Επίσης ορίζουμε πως η λειτουργία που εκτελείται σε αυτήν την συνάρτηση είναι η COP_DECRYPT. Τέλος, καλούμε τον οδηγό να κάνει την αποκρυπτογράφηση μέσω της ioctl.

main():

```
int main(int argc, char *argv[])
{
   int sd, port;
   ssize_t n;
   char buf[1000];
   char *hostname;
   struct hostent *hp;
   struct sockaddr_in sa;
   struct{
       unsigned char   in[DATA_SIZE],
                       encrypted[DATA_SIZE],
                       decrypted[DATA_SIZE];
   } data;


   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");


   /* Set key value */
   char str[16];
   sprintf(str, "oslab");
   crypto_key(str, strlen(str));
   /* Initialize cryptodev */
   cryptodev_init();


   /*
    * 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);
   }
   */

   while(1){

       bzero(buf, sizeof(buf));

       //Prompt To
       printf("To server: ");
       fgets(buf, sizeof(buf), stdin);

       buf[sizeof(buf) - 1] = '\0';


       for(int i = 0; i < DATA_SIZE; i++){
           data.in[i] = buf[i];
           if(i >= strlen(buf))
                   break;
       }

       /* Encrypt outbox message */
       encrypt(data.in, data.encrypted, sizeof(data.in));

       if (insist_write(sd, data.encrypted, strlen(data.encrypted)) !=
strlen(data.encrypted)) {
           perror("write");
           exit(1);

       }

       printf("Waiting for response...\n");
```

```
        /* Read answer and write it to standard output */
        for (;;) {
            n = read(sd, buf, sizeof(buf));

            if (n < 0) {
                perror("read");
                exit(1);
            }

            if (n <= 0)
                break;


            break;
        }

        for(int i = 0; i < DATA_SIZE; i++)
            data.encrypted[i] = buf[i];

        /* Decrypt outbox message for check */
        decrypt(data.encrypted, data.decrypted, sizeof(data.encrypted));

        //Prompt From Server
        printf("From server: %s",data.decrypted);
        if ((strncmp(data.decrypted, "exit", 4)) == 0) {
            printf("Client Exit...\n");
            break;
        }
    }

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

Η main() συνάρτηση του client εγκαθιστά την σύνδεση με το απομακρυσμένα socket του server και στο εσωτερικό της υλοποιείται η επικοινωνία με τον server. Εν ολίγοις, ο client στέλνει δεδομένα στον server και περιμένει την απάντησή του προτού ξαναστείλει, ενώ ιδιαίτερη προσοχή θέλει το καθάρισμα-άδειασμα του buffer που χρησιμοποιούμε για την ανάγνωση και εγγραφή δεδομένων. Στο κομμάτι των κρυπτογραφημένων δεδομένων, η main() καλεί τις προαναφερθείσες συναρτήσεις για την αρχικοποίηση του cryptodev, τις κλήσεις ioctl() για την έναρξη του session, τη διαμόρφωση του key (key=oslab εν προκειμένω) και του initialization vector (που στην περίπτωσή μας ορίζουμε ως 0 σε όλες τις θέσεις) καθώς και την κρυπτογράφηση/αποκρυπτογράφηση δεδομένων.

Στο socket-server.c:
Οι συναρτήσεις cryptodev_init(), crypto_key(), encrypt(), και decrypt() χρησιμοποιούνται και εδώ αυτούσιες και για ακριβώς τους ίδιους λόγους όπως και στον client, επομένως δεν τις επαναλαμβάνουμε στην αναφορά. Ακολουθεί η main που είναι κάπως διαφορετική, αν και αντίστοιχη:

main():

```
int main(void)
{
   unsigned char buf[1000];
   char addrstr[INET_ADDRSTRLEN];
   int sd, newsd;
   ssize_t n;
   socklen_t len;
   struct sockaddr_in sa;
   struct{
       unsigned char   in[DATA_SIZE],
                       encrypted[DATA_SIZE],
                       decrypted[DATA_SIZE];
   } data;


   /* 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);
   }

   /* Set key value */
   char str[16];
   sprintf(str, "oslab");
   crypto_key(str, strlen(str));
   /* Initialize cryptodev */
   cryptodev_init();

   /* 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 (;;) {
            bzero(buf, sizeof(buf));
            n = read(newsd, buf, sizeof(buf));
            if (n <= 0) {
                if (n < 0)
                    perror("read from remote peer failed");
                else
                    fprintf(stderr, "Peer went away\n");
                break;
            }

            for(int i = 0; i < DATA_SIZE; i++){
                data.encrypted[i] = buf[i];
                if(i >= strlen(buf))
                    break;

            }

            decrypt(data.encrypted, data.decrypted, sizeof(data.encrypted));

            //Prompt From client:
            printf("From client: %s", data.decrypted);


            if ((strncmp(data.decrypted, "exit", 4)) == 0) {
                printf("Client Exit...\n");

                break;
            }
            bzero(buf, sizeof(buf));

            //Prompt To client:
            printf("To client: ");

            fgets(buf, sizeof(buf), stdin);

            buf[sizeof(buf) - 1] = '\0';


            for(int i = 0; i < DATA_SIZE; i++)
                data.in[i] = buf[i];

            encrypt(data.in, data.encrypted, sizeof(data.in));

            if (insist_write(newsd, data.encrypted, strlen(data.encrypted)) !=
strlen(data.encrypted)) {
                perror("write to remote peer failed");
                break;
            }
            printf("Waiting for response...\n");

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

    /* This will never happen */
```

```
    return 1;
}
```

Όπως και στην main() συνάρτηση του client, στον server υλοποιούμε την μέσω socket επικοινωνία με κάποιον client. Προτού όμως δεχθεί κάποιον client στη main() ο server αναμένει νέες συνδέσεις θέτοντας σε κατάσταση listen το socket που έχει οριστεί. Κατά τα άλλα η λειτουργία της main() είναι ανάλογη της main() του client.

Στο socket-common.h:
Χρησιμοποιήσαμε αυτό το αρχείο για να ορίσουμε τα struct sess και cryp, τα iv και key, όπως και τον περιγραφητή αρχείου cfd, χρησιμοποιώντας και τα απαραίτητα define. Ο κώδικας που προστέθηκε εδώ είναι οι εξής γραμμές:

```
#define BLOCK_SIZE      16
#define KEY_SIZE  16  /* AES128 */

struct session_op sess;
struct crypt_op cryp;
__u8 iv[BLOCK_SIZE];
__u8 key[KEY_SIZE];
int cfd;
```

Ζητούμενο Ζ3 (front-end)

Στο 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;
        struct crypto_open_file *crof;
```

```
        struct crypto_device *crdev;
        unsigned int *syscall_type;
        int *host_fd;
        struct virtqueue *vq;

        //declare syscall_type and host_fd scatterlist
        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;

        vq = crdev->vq;

        /**
         * 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[0] = &syscall_type_sg;
        sg_init_one(&host_fd_sg, &crof->host_fd, sizeof(int));
        sgs[1] = &host_fd_sg;


        if(down_interruptible(&crdev->sem))
                return -ERESTARTSYS;
        err = virtqueue_add_sgs(vq, sgs, 1, 1, &syscall_type_sg, GFP_ATOMIC);
        virtqueue_kick(vq);


        /**
         * Wait for the host to process our data.
         **/
        /* ?? */
        while (virtqueue_get_buf(vq, &len) == NULL)
                /* do nothing */;

        debug("host fd:%d, len: %d", crof->host_fd, len);


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

```
        kfree(syscall_type);

        /* If host failed to open() return -ENODEV. */
        /* ?? */
        debug("I am before host failed to open()");
        if(crof->host_fd < 0){
                ret = -ENODEV;
                goto fail;
        }

        //debug("I am before exit");
        return ret;

fail:
        debug("Leaving with ret: %d\n", ret);
        return ret;
}

static int crypto_chrdev_release(struct inode *inode, struct file *filp)
{
        int ret = 0, err;
        unsigned int len;
        struct crypto_open_file *crof = filp->private_data;
        struct crypto_device *crdev = crof->crdev;
        unsigned int *syscall_type;
        //declare syscall_type and host_fd scatterlist
        struct scatterlist syscall_type_sg, host_fd_sg, *sgs[2];
        struct virtqueue *vq;


        debug("Entering");

        syscall_type = kzalloc(sizeof(*syscall_type), GFP_KERNEL);
        *syscall_type = VIRTIO_CRYPTODEV_SYSCALL_CLOSE;
        vq = crdev->vq;


        /**
         * Send data to the host.
         **/
        /* ?? */
        sg_init_one(&syscall_type_sg, syscall_type, sizeof(*syscall_type));
        sgs[0] = &syscall_type_sg;
        sg_init_one(&host_fd_sg, &crof->host_fd, sizeof(int));
        sgs[1] = &host_fd_sg;

        debug("Before lock");

        if(down_interruptible(&crdev->sem))
                return -ERESTARTSYS;
        debug("Before virtqueue_add_sgs close");
        err = virtqueue_add_sgs(vq, sgs, 2, 0, &syscall_type_sg, GFP_ATOMIC);
        virtqueue_kick(vq);

        /**
         * Wait for the host to process our data.
         **/
        /* ?? */
        debug("Before backend close");
        while (virtqueue_get_buf(vq, &len) == NULL)
                /* do nothing */;


        up(&crdev->sem);
        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, *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, ses_id_sg, crypt_op_sg,
src_sg, iv_sg, dst_sg,
                                                host_return_val_sg, *sgs[8];
        unsigned int num_out = 0, num_in = 0, *ioctl_cmd, *syscall_type, len;
        void *key_userspace, *session_id_userspace, *src_userspace, *iv_userspace,
*dst_userspace;

        struct session_op *sess, *sess_userspace = NULL;
        struct crypt_op *cryp, *cryp_userspace = NULL;


        debug("Entering");

        /**
         * Allocate all data that will be sent to the host.
         **/

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

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

        host_return_val = kzalloc(sizeof(int), GFP_KERNEL);
        *host_return_val = -1;

        /**
         *  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(int));
        sgs[num_out++] = &host_fd_sg;
        sg_init_one(&ioctl_cmd_sg, ioctl_cmd, sizeof(unsigned int));
        sgs[num_out++] = &ioctl_cmd_sg;
        /* ?? */

        /**
         *  Add all the cmd specific sg lists.
         **/
        switch (cmd) {
        case CIOCGSESSION:
                debug("CIOCGSESSION");

                sess_userspace = (struct session_op *) arg;
                sess = kzalloc(sizeof(struct session_op), GFP_KERNEL);
                if(copy_from_user(sess, sess_userspace, sizeof(struct
session_op))){
                        debug("Copy session_op from user failed!");
                        return -EFAULT;
                }

                key_userspace = sess->key;
                sess->key = kzalloc(sess->keylen * sizeof(unsigned char),
GFP_KERNEL);
                if(copy_from_user(sess->key, key_userspace, sess->keylen *
sizeof(unsigned char))){
```

```
                        debug("Copy session key from user failed!");
                        return -EFAULT;
                }

                //session_key_sg
                sg_init_one(&session_key_sg, sess->key, sess->keylen *
sizeof(unsigned char));
                sgs[num_out++] = &session_key_sg;

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

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

                /**
                 * Wait for the host to process our data.
                 **/
                /* ?? */
                /* ?? Lock ?? */
                if(down_interruptible(&crdev->sem))
                        return -ERESTARTSYS;
                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 */;

                sess->key = key_userspace;
                if(copy_to_user(sess_userspace, sess, sizeof(struct session_op))){
                        debug("Copy session_op to user failed!");
                        return -EFAULT;
                }

                up(&crdev->sem);

                break;


        case CIOCFSESSION:
                debug("CIOCFSESSION");


                session_id_userspace = kzalloc(sizeof(__u32), GFP_KERNEL);
                if(copy_from_user(session_id_userspace, (void *) arg,
sizeof(__u32))){
                        debug("Copy session ses from user failed!");
                        return -EFAULT;
                }

                //ses_id_sg
                sg_init_one(&ses_id_sg, session_id_userspace, sizeof(__u32));
                sgs[num_out++] = &ses_id_sg;

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

                /**
                 * Wait for the host to process our data.
                 **/
                /* ?? */
                /* ?? Lock ?? */
                if(down_interruptible(&crdev->sem))
                        return -ERESTARTSYS;
                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->sem);

                break;

        case CIOCCRYPT:
                debug("CIOCCRYPT");

                cryp_userspace = (struct crypt_op *) arg;
                cryp = kzalloc(sizeof(struct crypt_op), GFP_KERNEL);
                if(copy_from_user(cryp, cryp_userspace, sizeof(struct crypt_op))){
                        debug("Copy crypt_op from user failed!");
                        return -EFAULT;
                }

                src_userspace = cryp->src;
                cryp->src = kzalloc(cryp->len * sizeof(unsigned char),
GFP_KERNEL);
                if(copy_from_user(cryp->src, src_userspace, cryp->len *
sizeof(unsigned char))){
                        debug("Copy cryp src from user failed!");
                        return -EFAULT;
                }

                iv_userspace = cryp->iv;
                cryp->iv = kzalloc(16 * sizeof(unsigned char), GFP_KERNEL);
                if(copy_from_user(cryp->iv, iv_userspace, 16 * sizeof(unsigned
char))){
                        debug("Copy crypt iv from user failed!");
                        return -EFAULT;
                }

                dst_userspace = cryp->dst;
                cryp->dst = kzalloc(cryp->len * sizeof(unsigned char),
GFP_KERNEL);
                if(copy_from_user(cryp->dst, dst_userspace, cryp->len *
sizeof(unsigned char))){
                        debug("Copy crypt dst from user failed!");
                        return -EFAULT;
                }

                //crypt_op_sg
                sg_init_one(&crypt_op_sg, cryp, sizeof(struct crypt_op));
                sgs[num_out++] = &crypt_op_sg;

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

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

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

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

```
                /**
                 * Wait for the host to process our data.
                 **/
                /* ?? */
                /* ?? Lock ?? */
                if(down_interruptible(&crdev->sem))
                        return -ERESTARTSYS;
                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 */;

                cryp->dst = dst_userspace;
                if(copy_to_user(dst_userspace, cryp->dst, cryp->len *
sizeof(unsigned char) != 0)){
                        debug("Copy cryp->dst to user failed!");
                        up(&crdev->sem);
                        return -EFAULT;
                }


                up(&crdev->sem);

                break;

        default:
                debug("Unsupported ioctl command");

                break;
        }

        ret = (long) *host_return_val;

        kfree(syscall_type);

        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");
}
```

Το αρχείο crypto-chrdev.c αποτελεί την «καρδιά» του οδηγού που ετοιμάζουμε και που είναι ουσιαστικά το front-end τμήμα της συσκευής μας. Σε αυτό ορίζονται οι μέθοδοι του οδηγού όπως και στην δεύτερη εργαστηριακή άσκηση, μόνο που σε αυτήν την περίπτωση χρησιμοποιούμε την μέθοδο ioctl() στην οποία γίνεται η ανταλλαγή δεδομένων μεταξύ guest και host. Ακολουθώντας τη σειρά κλήσεων των μεθόδων κατά την εισαγωγή του module στον πυρήνα και την δοκιμή ενός παραδείγματος έχουμε τις εξής συναρτήσεις. Μετά την init() την οποία δεν έχουμε αλλάξει καλείται η open() κατά την οποία συσχετίζεται μια virtio συσκευή με μία συσκευή χαρακτήρα και εισάγουμε τα απαραίτητα δεδομένα στην scatter-gather list ώστε να προωθηθούν στο backend. Αντίστοιχη λογική έχει και η release() μέθοδος. Στην ioctl() διακρίνουμε τις περιπτώσεις των system calls που ζητά ο guest να πραγματοποιηθούν καθώς και των παραμέτρων του cryptodev αναλόγως της λειτουργίας που θέλουμε να εκτελέσουμε. Δεδομένου ότι ο οδηγός επεξεργάζεται δεδομένα από τον χώρο του χρήστη κάνουμε χρήση των συναρτήσεων copy_from_user() και copy_to_user() για την ασφαλή αντιγραφή των δεδομένων. Κάθε στοιχείο που θέλουμε να περάσουμε στο back-end το προσθέτουμε ως στοιχείο scatter-gather (sg) στην λίστα αρχικοποιώντας το με την sg_init_one() και προσθέτοντας το σε μια δομή scatterlist μέσω ανάθεσης σε κάποια θέση η οποία εξαρτάται από τα στοιχεία τα οποία που περιμένουμε είτε απλώς να διαβάσει (num_out) ή και να εγγράψει (num_in) το back-end (π.χ. `sg_init_one(&syscall_type_sg, syscall_type, sizeof(*syscall_type));` `sgs[num_out++] = &syscall_type_sg;`).

Στο crypto.h:

```
/**
* Global driver data.
**/
struct crypto_driver_data {
   /* The list of the devices we are handling. */
   struct list_head devs;

   /* The minor number that we give to the next device. */
   unsigned int next_minor;

   spinlock_t lock;
};
extern struct crypto_driver_data crdrvdata;


/**
* Device info.
**/
struct crypto_device {
   /* Next crypto device in the list, head is in the crdrvdata struct */
   struct list_head list;

   /* The virtio device we are associated with. */
   struct virtio_device *vdev;

   struct virtqueue *vq;

   /* ?? Lock ?? */
   struct semaphore sem;

   /* The minor number of the device. */
   unsigned int minor;
};
```

Στο υπάρχον αρχείο προσθέτουμε στις πιο πάνω δομές ένα πεδίο spinlock και semaphore αντίστοιχα για τη διαχείριση της εισαγωγής των αντίστοιχων συναρτήσεων στο critical section.

Στο crypto-module.c:

```
static int virtcons_probe(struct virtio_device *vdev)
{
   int ret = 0;
   struct crypto_device *crdev;

   debug("Entering");

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

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

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

   /* Other initializations. */
   /* ?? */
   sema_init(&crdev->sem, 1);

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

   debug("Leaving");

out:
   return ret;
}
```

Στο υπάρχον αρχείο προσθέτουμε στην συνάρτηση virtcons_probe() την αρχικοποίηση του semaphore που ορίσαμε στο crypto.h για την virtio συσκευή.

Στο virtio-cryptodev.c:

Όλες οι προσθήκες που έγιναν ήταν στην συνάρτηση vq_handle_output(), η οποία ακολουθεί:

vq_handle_output():

```
static void vq_handle_output(VirtIODevice *vdev, VirtQueue *vq)
   {
       VirtQueueElement *elem;
       unsigned int *syscall_type, *ioctl_cmd;
       int *host_fd;
       struct session_op *sess;
       struct crypt_op *cryp;
       int *host_return_val;

       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:%d", *host_fd);

           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("Error closing /dev/crypto");

           break;

       case VIRTIO_CRYPTODEV_SYSCALL_TYPE_IOCTL:
           DEBUG("VIRTIO_CRYPTODEV_SYSCALL_TYPE_IOCTL");
           /* ?? */
           host_fd = elem->out_sg[1].iov_base;
           ioctl_cmd = elem->out_sg[2].iov_base;


           switch (*ioctl_cmd){
           case CIOCGSESSION:
               DEBUG("CIOCGSESSION");

               sess = (struct session_op *) elem->in_sg[0].iov_base;
               sess->key = (unsigned char*) elem->out_sg[3].iov_base;

               host_return_val = elem->in_sg[1].iov_base;
               *host_return_val = ioctl(*host_fd, CIOCGSESSION, sess);
               if (*host_return_val) {
                   perror("ioctl(CIOCGSESSION) error");
               }
               break;
```

```
            case CIOCCRYPT:
                DEBUG("CIOCCRYPT");

                cryp = (struct crypt_op *) elem->out_sg[3].iov_base;
                cryp->src = elem->out_sg[4].iov_base;
                cryp->iv = elem->out_sg[5].iov_base;
                cryp->dst = elem->in_sg[0].iov_base;

                host_return_val = elem->in_sg[1].iov_base;

                *host_return_val = ioctl(*host_fd, CIOCCRYPT, cryp);
                if (*host_return_val) {
                    perror("ioctl(CIOCCRYPT)");
                    DEBUG("host_return_val");

                }
                printf("Cryp_src = %s\n", cryp->src);
                printf("Cryp_dst = %s\n", cryp->dst);
                printf("host return val is %d\n", *host_return_val);
                break;

            case CIOCFSESSION:
                DEBUG("CIOCFSESSION");

                unsigned int sess_id = * (unsigned int*) elem->out_sg[3].iov_base;

                host_return_val = elem->in_sg[0].iov_base;
                *host_return_val = ioctl(*host_fd, CIOCFSESSION, sess_id);

                if (*host_return_val) {
                    perror("ioctl(CIOCFSESSION)");
                }
                break;

            default:
                DEBUG("Unknown ioctl command");

            }

            break;

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

        DEBUG("LEAVING");
        virtqueue_push(vq, elem, 0);
        virtio_notify(vdev, vq);
        g_free(elem);
    }
```

Σε αυτήν την συνάρτηση υλοποιούνται όλες οι λειτουργίες του οδηγού, από την μεριά του backend. Αρχικά, παίρνουμε το πρώτο στοιχείο από την virtqueue για να αρχίσουμε την επεξεργασία του, αν αυτό υπάρχει. Το στοιχείο αυτό περιέχει τα scatter gather lists που έχει στείλει το frotend στο backend για ανάγνωση και επεξεργασία. Εμείς ακολουθήσαμε την πρόταση που δόθηκε στα σχήματα της σελίδας 14 στον Οδηγό της Τρίτης Εργαστηριακής, με όσα πεδία έχουν Read flag να αντιστοιχούν στο out_sg (sg που παραλαμβάνει το backend για ανάγνωση) και όσα έχουν Write flag να αντιστοιχούν στο in_sg (sg που στέλνει το backend στο frontend με καινούργιες πληροφορίες

(επεξεργασμένα)). Οι θέσεις στα out_sg και in_sg του κάθε πεδίου είναι αντίστοιχες με την θέση που εμφανίζεται κάθε πεδίο στους πίνακες στον Οδηγό.

Συμφωνά με τα παραπάνω λοιπόν, παραλαμβάνω την πληροφορία σχετικά με το τι κλήση συστήματος έχω από την πρώτη θέση του out_sg. Εάν είναι open, καλώ την συνάρτηση open για τον οδηγό, και φροντίζω να γραφτεί το fd που θα προκύψει ως αποτέλεσμα της open στην πρώτη θέση του in_sg (αυτό το κάνω συσχετίζοντας τον pointer host_fd, στο οποίο θα γραφεί το αποτέλεσμα, με τον pointer της συγκεκριμένης θέσης του in_sg - την ίδια τακτική ακολουθώ κάθε φορά που θέλω να εισάγω πληροφορία σε κάποια θέση του in_sg). Εάν είναι close, παίρνω το fd από την δεύτερη θέση του out_sg, και καλώ την close με αυτό ως όρισμα.

Εάν είναι ioctl, παίρνω το fd (host_fd) από την δεύτερη θέση του out_sg και το ioctl command (ioctl_cmd) από την τρίτη θέση του out_sg. Επίσης ορίζω τα structs sess και cryp που είναι τύπου session_op και crypt_op αντιστοίχως, όπως και το μέγεθος του κλειδιού (16) και το host_return_val. Εάν το ioctl_cmd είναι το CIOCGSESSION, τότε συσχετίζω το sess με το την πρώτη θέση του in_sg, ενώ παίρνω το κλειδί από την τέταρτη θέση του out_sg. Καλώ την ioctl για το sess (με λειτουργία CIOCGSESSION), και τοποθετώ το αποτέλεσμα στην δεύτερη θέση του in_sg. Εάν το ioctl_cmd είναι το CIOCCRYPT, τότε παίρνω το cryp από την τέταρτη θέση του out_sg, το src από την πέμπτη και το iv από την έκτη θέση, ενώ συσχετίζω το dst με την πρώτη θέση του in_sg.Ύστερα καλώ την ioctl για το cryp (με λειτουργία CIOCCRYPT) , της οποίας το αποτέλεσμα αποθηκεύεται στην δεύτερη θέση του in_sg. Εάν το ioctl_cmd είναι το CIOCFSESSION, παίρνω το sess_id από την τέταρτη θέση του out_sg, και καλώ την ioctl για το sess_id με λειτουργία το CIOCFSESSION, φροντίζοντας να γραφτεί το αποτέλεσμα στην πρώτη θέση του in_sg. Εάν το ioctl_cmd είναι ο,τιδήποτε άλλο δεν κάνουμε κάτι και ειδοποιούμε για άγνωστη εντολή, όπως και αν το syscall_type είναι κάτι άλλο εκτός από open, close, ή ioctl.

Τέλος, βάζουμε το επεξεργασμένο στοιχείο πίσω στο virtqueue και ειδοποιούμε το frontend για αυτό, ενώ ελευθερώνουμε και τον χώρο μνήμης που καταλάμβανε.