ΕΡΓΑΣΤΗΡΙΟ ΛΕΙΤΟΥΡΓΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

3η εργαστηριακή άσκηση

Γουλιάμου Μαρία-Έθελ

Ντούρος Ευάγγελος

Unencrypted chat

Σκοπός εδώ είναι να φτιάξουμε μια αμφίδρομη επικοινωνία τύπου chat μεταξύ **δύο ομότιμων (peers)**, συμβατικά ενός server και ενός client.

Οπως φαίνεται και στον κώδικά τους, το βασικό κομμάτι που υλοποιεί την επικοινωνία (διαβασμα και γράψιμο) είναι κοινό και για τους δύο. Το χαρακτηριστικό τους κομμάτι είναι ο τρόπος με τον οποίο μπορούν οι peers να διαβάζουν απο δύο file descriptors (stdin για είσοδο από χρήστη και socket για είσοδο από τον peer) και να ενεργούν ανάλογα με το ποιός έχει πρώτος δεδομένα. Αυτό επιτυγχάνεται με χρήση της συνάρτησης **poll()**.

server.c

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

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

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

#include "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;
}
```

```c
int main(int argc, char **argv)
{
    char buf[100];
    char addrstr[INET_ADDRSTRLEN];
    int sd, newsd;
    struct sockaddr_in sa;
    socklen_t len;
    ssize_t n;
    struct pollfd pfds[2];

    /*Make sure a broken connection doesn't kill us*/
    signal(SIGPIPE, SIG_IGN);

    /*Create TCP/IP socket, used as the main chat channel*/
    if ((sd = socket(PF_INET, SOCK_STREAM, 0)) < 0) { //protocol family IPv4, TCP
        perror("socket");
        exit(1);
    }
    fprintf(stderr, "Created TCP socket\n");

    /*Bind to a well-known port*/
    memset(&sa, 0, sizeof(sa));  //is used to zero sin_zero[8]
    sa.sin_family = AF_INET;    //address family IPv4
    sa.sin_port = htons(TCP_PORT); //convert to network byte order
    //sa.sin_addr.s_addr = htonl(INADDR_ANY); //binds the socket to all available
interfaces
    sa.sin_addr.s_addr = inet_addr("127.0.0.1"); //for localhost
    if (bind(sd, (struct sockaddr *)&sa, sizeof(sa)) < 0) {
        perror("bind");
        exit(1);
    }
    fprintf(stderr, "Bound TCP socket to port %d\n", TCP_PORT);

    /* Listen for incoming connections */
    if (listen(sd, TCP_BACKLOG) < 0) {
        perror("listen");
        exit(1);
    }

    /*Loop forever, accepting 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 (;;) {
            //poll 0 and neasd to see which has data first
            pfds[0].fd = 0;
            pfds[0].events = POLLIN;

            pfds[1].fd = newsd;
            pfds[1].events = POLLIN;
```

```
            poll(pfds, 2, 0);
            if (pfds[0].revents & POLLIN) {
                n = read(0, buf, sizeof(buf));
                if (n < 0) {
                    perror("[server] read from stdin");
                    exit(1);
                }
                if (n == 0)  //EOF??
                    break;

                if (insist_write(newsd, buf, n) != n) {
                    perror("[server] write to peer");
                    exit(1);
                }
            }
            else if (pfds[1].revents & POLLIN) {
                n = read(newsd, buf, sizeof(buf));
                if (n <= 0) {
                    if (n < 0)
                        perror("[server] read from peer");
                    else
                        fprintf(stderr, "[server] peer went away\n");
                    break;
                }

                if (insist_write(1, buf, n) != n) {
                    perror("[server] write to stdout");
                    break;
                }
            }
        }
        /* Make sure we don't leak open files */
        if (close(newsd) < 0)
            perror("close");
    }

    /*Unreachable*/
    fprintf(stderr, "Reached unreachable point!\n");
    return 1;
}
```

client.c

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

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

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

#include "common.h"

/* Insist until all of the data has been written */
```

```c
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;
    char *hostname;
    char buf[100];
    struct hostent *hp;
    struct sockaddr_in sa;
    ssize_t n;
    struct pollfd pfds[2];

    if (argc != 3) {
        fprintf(stderr, "Usage: %s hostname port\n", argv[0]);
        exit(1);
    }
    hostname = argv[1];
    port = atoi(argv[2]);

    /* Create TCP/IP socket, used as the 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");

    for (;;) {
        //poll 0 and neasd to see which has data first
        pfds[0].fd = 0;
        pfds[0].events = POLLIN;

        pfds[1].fd = sd;
        pfds[1].events = POLLIN;
```

```
        poll(pfds, 2, 0);
        if (pfds[0].revents & POLLIN) {
            n = read(0, buf, sizeof(buf));
            if (n < 0) {
                perror("[client] read from client");
                exit(1);
            }
            if (n == 0)  //EOF??
                break;

            if (insist_write(sd, buf, n) != n) {
                perror("[client] write to peer");
                exit(1);
            }
        }
        else if (pfds[1].revents & POLLIN) {
            n = read(sd, buf, sizeof(buf));
            if (n <= 0) {
                if (n < 0)
                    perror("[client] read from peer");
                else
                    fprintf(stderr, "[client] peer went away...exiting\n");
                break;
            }

            if (insist_write(1, buf, n) != n) {
                perror("[client] write to stdout");
                exit(1);
            }
        }
    }

    return 0;
}
```

common.h

```
#ifndef _SOCKET_COMMON_H
#define _SOCKET_COMMON_H

/* Compile-time options */
#define TCP_PORT    35001
#define TCP_BACKLOG 5   //max queue of clients length

#endif /* _SOCKET_COMMON_H */
```

Τρέχουμε τον server με

```
$ ./server
```

Και τον client, σε διαφορετικό τερματικό με

```
$ ./client 127.0.0.1 35001
```

Καθώς έχουμε εγκαταστήσει την επικοινωνία μπορούμε, σε διαφορετικό τερματικό, να "ακούμε" την κίνηση καθώς δεν έχουμε εφαρμόσει κάποια κρυπτογράφηση στα δεδομένα ακόμη.

```
$ sudo tcpdump -A -i lo tcp -nnn -XXX -vvv
```

Encrypted chat

Εδώ ο στόχος ήταν τα μηνύματα που ανταλλάσσουν ο server και ο client να είναι κρυπτογραφημένα. Για αυτόν τον λόγο κάναμε χρήση του **cryptodev-linux**, έναν driver για κρυπρογραφικές συσκευές. Στην περίπτωσή μας, που δεν έχουμε κάποια συσκευή κρυπτογράφησης, η κρυπτογράφηση θα γίνεται από λογισμικό.

Οι αλλαγές στον κώδικα φαίνονται παρακάτω. Απλώς προστέθηκαν κλήσεις των συναρτήσεων **encrypt()** και **decrypt()**, μετά το διάβασμα από το πληκτρολόγιο και από το socket αντίστοιχα.

encrypt()

```
int encrypt(int cfd)
{
        int i;
        struct crypt_op cryp;
        struct {
                unsigned char   in[DATA_SIZE],
                                encrypted[DATA_SIZE],
                                iv[BLOCK_SIZE];
        } data;

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

        /*Encrypt data.in to data.encrypted*/
        cryp.ses = sess.ses;
        cryp.len = sizeof(data.in);
        cryp.src = buf;
        cryp.dst = data.encrypted;
        cryp.iv = inv;
        cryp.op = COP_ENCRYPT;

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

        i = 0;
        memset(buf, '\0', sizeof(buf));
        while(data.encrypted[i] != '\0'){
                buf[i] = data.encrypted[i];
                i++;
        }

        return 0;
}
```

decrypt()

```
int decrypt(int cfd){

        int i;
        struct crypt_op cryp;
        struct {
                unsigned char   in[DATA_SIZE],
                                decrypted[DATA_SIZE],
                                iv[BLOCK_SIZE];
        } data;

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

```
        /*Decrypt data.encrypted to data.decrypted*/
        cryp.ses = sess.ses;
        cryp.len = sizeof(data.in);
        cryp.src = buf;
        cryp.dst = data.decrypted;
        cryp.iv = inv;
        cryp.op = COP_DECRYPT;
        if (ioctl(cfd, CIOCCRYPT, &cryp)) {
                perror("ioctl(CIOCCRYPT)");
                return 1;
        }

        i = 0;
        memset(buf, '\0', sizeof(buf));
        while(data.decrypted[i] != '\0'){
                buf[i] = data.decrypted[i];
                i++;
        }

        return 0;
}
```

Αρχικά φορτώνουμε το module του driver στον πυρήνα

```
# /sbin/insmod ./cryptodev.ko
```

Ύστερα τρέχουμε τον server με

```
$ ./crypto-server
```

Και τον client, σε διαφορετικό τερματικό με

```
$ ./crypto-client 127.0.0.1 35001
```

Τώρα εάν, ομοίως με πριν, "ακούσουμε" την κίνηση στο δίκτυο θα δούμε μόνο κρυπτογραφημένα μηνύματα.

VirtIO-cryptodev

Σκοπός εδώ είναι να φτιάξουμε έναν driver για virtual machine ο οποίος να δίνει τη δυνατότητα χρήσης κρυπτογραφικών συσκευών, μέσα από εικονικά περιβάλλοντα. Για να επιτευχθεί αυτό θα πρέπει να φτιαχτεί ο driver για τον πυρήνα του guest μηχανήματος (frontend) αλλά και να γίνει επέκταση του πηγαίου κώδικα του QEMU φτιάχνοντας έτσι ένα εικονικό hardware (backend).

Γίνεται χρήση του προτύπου VirtIO για την μεταφορά δεδομένων από τον πυρηνα του guest στο QEMU.

Frontend

Το κύριο κομμάτι της υλοποίησης εδώ, ήταν η σωστή εφαρμογή του προτύπου VirtIO. Η βασική δυσκολία που αντιμετωπίστηκε ήταν η σωστή μεταφορά των structs από το userspace στον kernel. Συγκεκριμένα με την εντολή copy_from_user κάναμε αντιγραφή ενός struct αλλά τα πεδία key, iv και src έφερναν έναν δείκτη στα δεδομένα και όχι τα ίδια τα δεδομένα. Συνεπώς αυτά τα πεδία αντιγράφηκαν ξεχωριστά. Ιδιαίτερη προσοχή χρειάστηκε και στην εντολή copy_to_user ώστε να μεταφερθούν σωστά οι δείκτες για να δείχνουν στο userspace. Επισημαίνεται πως αυτοί οι δείκτες ενδέχεται μετά το copy_from_user να τροποποιήθηκαν και να δείχνουν σε kernelspace.

crypto-chrdev.c

```
#include <linux/cdev.h>
```

```c
#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 num_out, num_in, len;
        struct crypto_open_file *crof;
        struct crypto_device *crdev;
        unsigned int *syscall_type;
        int *host_fd;

    struct scatterlist syscall_type_sg, host_fd_sg, *sgs[2];
    unsigned long flags;

    num_out = 0;
    num_in = 0;

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

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

    err = virtqueue_add_sgs(crdev->vq, sgs, num_out, num_in, \
                                &syscall_type_sg, GFP_ATOMIC);
    virtqueue_kick(crdev->vq);
    while(virtqueue_get_buf(crdev->vq, &len) == NULL)
        ; //Do nothing

    spin_unlock_irqrestore(&crdev->lock, flags);

        /* If host failed to open() return -ENODEV. */
        if (crof->host_fd < 0) {
        debug("Host failed to open the crypto device");
        ret = -ENODEV;
    }
    debug("Host opened /dev/crypto file with fd = %d", crof->host_fd);
fail:
        debug("Leaving");
        return ret;
}

static int crypto_chrdev_release(struct inode *inode, struct file *filp)
{
        int ret = 0;
        struct crypto_open_file *crof = filp->private_data;
```

```
        struct crypto_device *crdev = crof->crdev;
        unsigned int *syscall_type;

    struct scatterlist syscall_type_sg, host_fd_to_close_sg, *sgs[2];
    unsigned int len, num_out, num_in;
    unsigned long flags;
    int err;

    num_out = 0;
    num_in = 0;

        debug("Entering");

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

        /**
         * Send data to the host.
         **/
     //now I send 2 scatterlists one for the syscall type and one for the fd
    sg_init_one(&syscall_type_sg, syscall_type, sizeof(syscall_type));
    sgs[num_out++] = &syscall_type_sg;
    sg_init_one(&host_fd_to_close_sg, &crof->host_fd, sizeof(crof->host_fd));
    sgs[num_out++] = &host_fd_to_close_sg;

    spin_lock_irqsave(&crdev->lock, flags);         //IS IRQSAVE NEEDED??
    err = 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)
        ; //Do nothing
    spin_unlock_irqrestore(&crdev->lock, flags);

    debug("Host closed /dev/crypto with fd = %d\n", crof->host_fd);
    /*
     * Check for close() failure?
     * Just add a host_ret pointer just like in ioctl methods below
     */

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

}

static long crypto_chrdev_ioctl(struct file *filp, unsigned int cmd,
                                unsigned long arg)
{
    int i;
        long ret = 0;
        int err;
        struct crypto_open_file *crof = filp->private_data;
        struct crypto_device *crdev = crof->crdev;
        struct virtqueue *vq = crdev->vq;
        struct scatterlist syscall_type_sg, ioctl_type_sg, output_msg_sg,
input_msg_sg,\
                        sess_sg, host_fd_sg, key_sg, host_ret_sg, ses_sg, cryp_sg,\
                        src_sg, dst_sg, iv_sg, op_sg,\
                            *sgs[14];  //correct size smaller??
        unsigned int num_out, num_in, len;
#define MSG_LEN 100
        unsigned char *output_msg, *input_msg, *key=NULL;
```

```
        unsigned int *syscall_type, *ioctl_type=NULL;
    long *host_ret = kmalloc(sizeof(long), GFP_KERNEL);
    struct session_op *sess=NULL;
    uint32_t *ses=NULL;
    struct crypt_op *cryp=NULL;
    unsigned char *src=NULL, *dst=NULL, *iv=NULL; //or uint8_t? Would it be the same?
    char bytes;
    unsigned long flags;
    num_out = 0;
        num_in = 0;
    *host_ret = -1; //by default lets have an error

    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;

        /**
         *  These are common to all ioctl commands.
         **/
        sg_init_one(&syscall_type_sg, syscall_type, sizeof(*syscall_type));
//out_sg[0]
        sgs[num_out++] = &syscall_type_sg;
    sg_init_one(&host_fd_sg, &crof->host_fd, sizeof(crof->host_fd));
//out_sg[1]
    sgs[num_out++] = &host_fd_sg;

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

        //define ioctl command
        ioctl_type = kzalloc(sizeof(*ioctl_type), GFP_KERNEL);
        *ioctl_type = VIRTIO_CRYPTODEV_IOCTL_CIOCGSESSION;

        //copy session_op struct from userspace
        //now sess is in kernel space but points to userspace
        sess = kzalloc(sizeof(*sess), GFP_KERNEL);
        if (copy_from_user(sess, (struct session_op *)arg, sizeof(*sess))) {
            debug("Copy session_op from user failed");
            ret = -EFAULT;
            goto out;
        }

        //copy key from userspace
        //sess has only a pointer to the string key
        //in order to have the actual key value I should copy that too
        key = kzalloc(sess->keylen, GFP_KERNEL);
        if (copy_from_user(key, sess->key, sess->keylen)) {
            debug("Copy key from user failed");
            ret = -EFAULT;
            goto out;
        }

        //send with R flag
        sg_init_one(&ioctl_type_sg, ioctl_type, sizeof(*ioctl_type));
//out_sg[2]
        sgs[num_out++] = &ioctl_type_sg;
        sg_init_one(&key_sg, key, sess->keylen);
//out_sg[3]
        sgs[num_out++] = &key_sg;
```

```
        //send with W flag (and R)
        sg_init_one(&sess_sg, sess, sizeof(*sess));
//in_sg[0]
        sgs[num_out + num_in++] = &sess_sg;
        sg_init_one(&host_ret_sg, host_ret, sizeof(*host_ret));
//in_sg[1]
        sgs[num_out + num_in++] = &host_ret_sg;
        break;

        case CIOCFSESSION:
                debug("CIOCFSESSION");

        //define ioctl command
        ioctl_type = kzalloc(sizeof(*ioctl_type), GFP_KERNEL);
        *ioctl_type = VIRTIO_CRYPTODEV_IOCTL_CIOCFSESSION;

        ses = kmalloc(sizeof(uint32_t), GFP_KERNEL);
        if (copy_from_user(ses, (uint32_t *)arg, sizeof(*ses))) {
            debug("Copy from user failed");
            ret = -EFAULT;
            goto out;
        }

        //send with R flag
        sg_init_one(&ioctl_type_sg, ioctl_type, sizeof(*ioctl_type));
//out_sg[2]
        sgs[num_out++] = &ioctl_type_sg;
        sg_init_one(&ses_sg, ses, sizeof(*ses));
//out_sg[3]
        sgs[num_out++] = &ses_sg;

        //send with W flag (and W)
        sg_init_one(&host_ret_sg, host_ret, sizeof(*host_ret));
//in_sg[0]
        sgs[num_out + num_in++] = &host_ret_sg;
                break;

        case CIOCCRYPT:
                debug("CIOCCRYPT");

        ioctl_type = kzalloc(sizeof(*ioctl_type), GFP_KERNEL);
        *ioctl_type = VIRTIO_CRYPTODEV_IOCTL_CIOCCRYPT;

        cryp = kmalloc(sizeof(*cryp), GFP_KERNEL);
        if (copy_from_user(cryp, (struct crypt_op *)arg, sizeof(*cryp))) {
            debug("Copy crypt_op from user failed");
            ret = -EFAULT;
            goto out;
        }

        src = kzalloc(cryp->len, GFP_KERNEL);
        if (copy_from_user(src, cryp->src, cryp->len)) {
            debug("Copy src from user failed");
            ret = -EFAULT;
            goto out;
        }

        iv = kzalloc(EALG_MAX_BLOCK_LEN, GFP_KERNEL);
        if (copy_from_user(iv, cryp->iv, EALG_MAX_BLOCK_LEN)) {
            debug("Copy iv from user failed");
            ret = -EFAULT;
            goto out;
        }
```

```
        dst = kzalloc(cryp->len, GFP_KERNEL);

        //send with R flag
        sg_init_one(&ioctl_type_sg, ioctl_type, sizeof(*ioctl_type));
//out_sg[2]
        sgs[num_out++] = &ioctl_type_sg;
        sg_init_one(&cryp_sg, cryp, sizeof(*cryp));
//out_sg[3]
        sgs[num_out++] = &cryp_sg;
        sg_init_one(&src_sg, src, cryp->len);
//out_sg[4]
        sgs[num_out++] = &src_sg;
        sg_init_one(&iv_sg, iv, EALG_MAX_BLOCK_LEN);
//out_sg[5]
        sgs[num_out++] = &iv_sg;

        //send with W flag (and R)
        sg_init_one(&host_ret_sg, host_ret, sizeof(*host_ret));
//in_sg[0]
        sgs[num_out + num_in++] = &host_ret_sg;
        sg_init_one(&dst_sg, dst, cryp->len);
//out_sg[1]
        sgs[num_out + num_in++] = &dst_sg;
        break;

    default:
                debug("Unsupported ioctl command");
                break;
        }

        /**
         * Wait for the host to process our data.
         **/
        spin_lock_irqsave(&crdev->lock, flags);
        err = virtqueue_add_sgs(vq, sgs, num_out, num_in,
                                &syscall_type_sg, GFP_ATOMIC);
        virtqueue_kick(vq);
        while (virtqueue_get_buf(vq, &len) == NULL)
                /* do nothing */;
    spin_unlock_irqrestore(&crdev->lock, flags);

    switch (cmd) {
    case CIOCGSESSION:
        //restore the pointer sess->key to original userspace address
        sess->key = ((struct session_op *) arg)->key;
        //and return the session to the userspace
        if (copy_to_user((struct session_op *) arg, sess, sizeof(*sess))) {
            debug("Copy to user failed!");
            ret = -EFAULT;
            goto out;
        }
        break;

    case CIOCFSESSION:
        //needed??????
        if (copy_to_user((uint32_t *)arg, ses, sizeof(*ses))) {
            debug("Copy to user failed!");
            ret = -EFAULT;
            goto out;
        }
        break;

    case CIOCCRYPT:
        //return dst data (encrypted/decrypted back to userspace)
        if (copy_to_user(((struct crypt_op *)arg)->dst, dst, cryp->len)) {
```

```
            debug("Copy to user failed!");
            ret = -EFAULT;
            goto out;
        }
        break;

    default:
                debug("Unsupported ioctl command (2nd)");
        break;
    }

    ret = *host_ret;
out:
    debug("Leaving ioctl with ret value %ld", ret);

    //FREE UP SPACE
    kfree(cryp);
    kfree(dst);
    kfree(src);
    kfree(iv);
    kfree(ioctl_type);
    kfree(ses);
    kfree(sess);
    kfree(key);
    kfree(output_msg);
    kfree(input_msg);
    kfree(syscall_type);
    kfree(host_ret);

        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");
}
```

Backend

Εδώ υπολοιήθηκε το εικονικό hardware. Στην ουσία το frontend κομμάτι στέλνει δεδομένα εδώ μέσω Virtqueues και στη συνέχεια το backend κομμάτι κάνει τις απαραίτητες κλήσεις συστήματος για να κάνει access στην πραγματική συσκευή του host μηχανήματος. Αφού ολοκληρωθεί αυτό, στέλνει τα νέα δεδομένα πίσω στο frontend κομμάτι. Αυτό με τη σειρά του ενημερώνει την userspace εφαρμογή.

virtio-cryptodev.c

```
#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_type;
    unsigned char *key;
    //unsigned char *output_msg, *input_msg,
    long *host_ret;
    struct session_op *sess;
    struct crypt_op *cryp;
    uint32_t *ses;
    unsigned char *src, *dst, *iv;
    //uint16_t *op;

    //fd_ptr should be head allocated
    int *fd_ptr, *fd_ptr_to_close;

    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");
        fd_ptr = elem->in_sg[0].iov_base;
        *fd_ptr = open(CRYPTODEV_FILENAME, O_RDWR);
        if (*fd_ptr < 0) {
            DEBUG("File does not exist");
        }
        printf("Opened /dev/crypto file with fd = %d\n", *fd_ptr);
        break;

    //better error check is needed
    case VIRTIO_CRYPTODEV_SYSCALL_TYPE_CLOSE:
        DEBUG("VIRTIO_CRYPTODEV_SYSCALL_TYPE_CLOSE");
        //fd_ptr could be used. No actual reason why fd_ptr_to_close is used
        fd_ptr_to_close = elem->out_sg[1].iov_base;
        if (close(*fd_ptr_to_close) < 0) {
            DEBUG("close() error");
        }
        else {
            printf("Closed /dev/crypto file with fd = %d\n", *fd_ptr_to_close);
        }
        break;
```

```
case VIRTIO_CRYPTODEV_SYSCALL_TYPE_IOCTL:
    DEBUG("VIRTIO_CRYPTODEV_SYSCALL_TYPE_IOCTL");
    fd_ptr = elem->out_sg[1].iov_base;
    ioctl_type = elem->out_sg[2].iov_base;
    switch (*ioctl_type) {
    case VIRTIO_CRYPTODEV_IOCTL_CIOCGSESSION:
        DEBUG("Entering CIOCGSESSION");
        key = elem->out_sg[3].iov_base;
        sess = elem->in_sg[0].iov_base;
        host_ret = elem->in_sg[1].iov_base;

        sess->key = key;
        *host_ret = ioctl(*fd_ptr, CIOCGSESSION, sess);
        if (*host_ret)
            perror("ioctl(CIOCGSESSION)");

        printf("The key is:\n");
        for(int i=0; i< sess->keylen;i++) {
            printf("%x", *(sess->key + i));
        }
        printf("\n");
        DEBUG("Leaving CIOCGSESSION");
        break;

    case VIRTIO_CRYPTODEV_IOCTL_CIOCFSESSION:
        DEBUG("Entering CIOCFSESSION");
        ses = elem->out_sg[3].iov_base;
        host_ret = elem->in_sg[0].iov_base;

        *host_ret = ioctl(*fd_ptr, CIOCFSESSION, ses);
        if (*host_ret)
            perror("ioctl(CIOCFSESSION)");

        DEBUG("Leaving CIOCFSESSION");
        break;

    case VIRTIO_CRYPTODEV_IOCTL_CIOCCRYPT:
        DEBUG("Entering CIOCCRYPT");
        cryp = elem->out_sg[3].iov_base;
        src = elem->out_sg[4].iov_base;
        iv = elem->out_sg[5].iov_base;
        host_ret = elem->in_sg[0].iov_base;
        dst = elem->in_sg[1].iov_base;

        cryp->src = src;
        cryp->dst = dst;
        cryp->iv = iv;
        *host_ret = ioctl(*fd_ptr, CIOCCRYPT, cryp);
        if (*host_ret)
            perror("ioctl(CIOCCRYPT)");

        printf("\n");
        DEBUG("Leaving CIOCCRYPT");
        break;

    default:
        DEBUG("Unsupported ioctl command!");
        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)
```