Εθνικό Μετσόβιο Πολυτεχνείο

Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

Εργαστήριο Λειτουργικών Συστημάτων 2021-2022

3η Εργαστηριακή Άσκηση

Ομάδα 51

Κωνσταντίνος Σιδέρης, Α.Μ.: 03118134

Ζητούμενα 1 & 2

Για την υλοποίηση αμφίδρομης επικοινωνίας (chat) μεταξύ δύο peers, server και client χρησιμοποιούμε TCP/IP. Επιλέγουμε πρωτόκολλο TCP διότι για εφαρμογές ανταλλαγής μηνυμάτων μας ενδιαφέρει περισσότερο η εγγύηση αποστολής όλων των μηνυμάτων και με τη σωστή σειρά παρά η ταχύτητα που προσφέρει το πρωτόκολλο UDP. Τα μηνύματα που ανταλλάσσονται κρυπτογραφούνται/αποκρυπτογραφούνται από τις συναρτήσεις encrypt_data και decrypt_data οι οποίες υλοποιούνται με την χρήση εντολών ioctl του driver cryptodev και είναι συμμετρικές για τον server και τον client.

encrypt_data

Η συνάρτηση δέχεται ως όρισμα τον file descriptor (cfd) της κρυπτογραφικής συσκευής που έχουμε ανοίξει και με την οποία έχουμε δημιουργήσει συνεδρία (session). Αρχικοποιούμε το struct crypt_op crypt και περνάμε σε αυτό τα δεδομένα κρυπτογράφησης όπως το αναγνωριστικό ses συνεδρίας με την συσκευή (cryp.ses = sess.ses), το μέγεθος μηνύματος (cryp.len = DATA_SIZE), τον πίνακα χαρακτήρων προέλευσης (cryp.src = buf), τον πίνακα χαρακτήρων προορισμού(cryp.dst = encrypted), το initialisation vector (cryp.iv = inv) και την εντολή που θέλουμε να εκτελέσουμε, στην περίπτωση μας κρυπτογράφηση (cryp.op = COP_ENCRYPT). Τέλος, καλούμε την ioctl με ορίσματα CIOCCRYPT, crypt και αντιγράφουμε το encrypted μήνυμα στον global πίνακα χαρακτήρων buf από τον οποίο μπορεί να τα διαβάσει το κύριο πρόγραμμα. να πρόβλημα που συναντήσαμε στην υλοποίηση ήταν ότι για συγκεκριμένες εισόδους (ανάλογα με το κλειδί της κρυπτογράφησης) γινόταν λάθος αποκρυπτογράφηση. Ο λόγος που συνέβαινε αυτό ήταν διότι κατά την κρυπτογράφηση αντιγράφαμε το μήνυμα μέχρι να συναντήσουμε ‘\0’ και συνεπώς για ορισμένα κρυπτογραφημένα μηνύματα που περιείχαν τον χαρακτήρα ‘\0’ δεν αντιγράφαμε ολόκληρο το μήνυμα. Ακολουθεί ο κώδικας της συνάρτησης encrypt_data:

```
int encrypt_data(int cfd)
{
    int i = 0;
    struct crypt_op cryp;
    unsigned char encrypted[DATA_SIZE];

    memset(encrypted, '\0', sizeof(encrypted));
    memset(&cryp, 0, sizeof(cryp));

    cryp.ses = sess.ses;
    cryp.len = DATA_SIZE;
    cryp.src = buf;
    cryp.dst = encrypted;
    cryp.iv = inv;
```

```
    cryp.op = COP_ENCRYPT;

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

    memset(buf, '\0', sizeof(buf));
    for(i = 0; i < DATA_SIZE; i++) {
        buf[i] = encrypted[i];
    }
    return 0;
}
```

decrypt_data

Η συνάρτηση, επίσης, δέχεται ως όρισμα τον file descriptor (cfd) της κρυπτογραφικής συσκευής. Ομοίως με την encrypt_data, αρχικοποιούμε το struct crypt_op crypt με διαφορά την εντολή που θέλουμε να εκτελέσουμε, σε αυτή την περίπτωση αποκρυπτογράφηση (cryp.op = COP_DECRYPT). Τέλος, καλούμε την ioctl με ορίσματα CIOCCRYPT, crypt και αντιγράφουμε το decrypted μήνυμα, μέχρι να συναντήσουμε χαρακτήρα '\0', στον global πίνακα χαρακτήρων buf από τον οποίο μπορεί να τα διαβάσει το κύριο πρόγραμμα. Ακολουθεί ο κώδικας της συνάρτησης decrypt_data:

```
int decrypt_data(int cfd)
{
    int i = 0;
    struct crypt_op cryp;
    unsigned char decrypted[DATA_SIZE];

    memset(decrypted, '\0', sizeof(decrypted));
    memset(&cryp, 0, sizeof(cryp));

    cryp.ses = sess.ses;
    cryp.len = DATA_SIZE;
    cryp.src = buf;
    cryp.dst = decrypted;
    cryp.iv = inv;
    cryp.op = COP_DECRYPT;

    if (ioctl(cfd, CIOCCRYPT, &cryp)) {
        perror("server ioctl(CIOCCRYPT): decrypt");
```

```
        return 1;
    }

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

socket-server.c

Δημιουργούμε ένα TCP/IP socket το οποίο κάνουμε bind με το port 35001 και το προετοιμάζουμε ώστε να μπορεί να δεχτεί (να “ακούει” για) εισερχόμενες συνδέσεις με την listen. Ο server περιμένει για νέες συνδέσεις από clients σε αέναο βρόχο. Δέχεται εισερχόμενες συνδέσεις με την accept, η οποία επιστρέφει ένα νέο socket (newsd) το οποίο χρησιμοποιείται για την επικοινωνία client-server. Το αρχικό socket δεν επηρεάζεται από την δημιουργία του νέου και επαναχρησιμοποιείται αφού τερματιστεί η επικοινωνία. Αφού κατοχυρωθεί η επικοινωνία, ανοίγουμε την συσκευή dev/crypto και αρχικοποιούμε το struct session_op sess κατάλληλα ώστε να δημιουργήσουμε νέα συνεδρία (session) με την συσκευή καλώντας την ioctl με όρισμα CIOCGSESSION. Στην συνέχεια, μπαίνουμε σε νέο αέναο βρόχο επικοινωνίας με τον client από τον οποίο μπορούμε να βγούμε εάν ο server ή ο client στείλουν τον χαρακτήρα ‘\0’, δηλαδή EOF. Για την αποστολή και λήψη μηνυμάτων δημιουργούμε ένα file descriptor set (FD_ZERO(&rfds)) στο οποίο προσθέτουμε τους file descriptors 0 (stdin) ώστε να διαβάζουμε μηνύματα που θέλει να στείλει ο server και το newsd ώστε να διαβάζουμε μηνύματα που στέλνει ο client. Παρακολουθούμε το fd set χρησιμοποιώντας την select και την FD_ISSET η οποία βρίσκει σε ποιον file descriptor του set έχουν γραφτεί δεδομένα. Εάν έχει γραφτεί νέο μήνυμα στο stdin του server, τα κρυπτογραφούμε με την encrypt_data και το γράφουμε στο socket με το οποίο επικοινωνούμε με τον client. Διαφορετικά, εάν έχει γραφτεί νέο μήνυμα στο socket με το οποίο επικοινωνούμε με τον client, το αποκρυπτογραφούμε με την decrypt_data και το εμφανίζουμε στο stdout του server. Ακολουθεί ο κώδικας της συνάρτησης main του αρχείου socket-server.c:

```
int main(int argc, char *argv[])
{
    char addrstr[INET_ADDRSTRLEN], *filename;
    int sd, newsd, retval, cfd;
    ssize_t n;
    socklen_t len;
    fd_set rfds;
    struct sockaddr_in sa;
```

```
if (argc > 2) {
    fprintf(stderr, "Incorrect arguements");
    exit(1);
}

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

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

/* Bind to a well-known port */
memset(&sa, 0, sizeof(sa));
sa.sin_family = AF_INET;
sa.sin_port = htons(TCP_PORT);
sa.sin_addr.s_addr = htonl(INADDR_ANY);
if (bind(sd, (struct sockaddr *)&sa, sizeof(sa)) < 0) {
    perror("bind");
    exit(1);
}
fprintf(stderr, "Bound TCP socket to port %d\n", TCP_PORT);

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

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

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

```
if (!inet_ntop(AF_INET, &sa.sin_addr, addrstr, sizeof(addrstr))) {
    perror("could not format IP address");
    exit(1);
}
fprintf(stderr, "Incoming connection from %s:%d\n",
    addrstr, ntohs(sa.sin_port));

filename = (argv[1] == NULL) ? "/dev/crypto" : argv[1];
cfd = open(filename, O_RDWR);
if (cfd < 0) {
    perror("server open(/dev/crypto)");
    exit(1);
}

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

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

    for (;;) {
        FD_ZERO(&rfds);
        FD_SET(0, &rfds);
        FD_SET(newsd, &rfds);
        retval = select(newsd+1, &rfds, NULL, NULL, NULL);
        if (retval < 0) {
            perror("server select");
            exit(1);
        }
        else if (FD_ISSET(0, &rfds)) {
            memset(buf, '\0', sizeof(buf));
            n = read(0, buf, sizeof(buf));

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

```
            if (n == 0) break;

            if (encrypt_data(cfd)) {
                perror("server encrypt");
                exit(1);
            }

            if (insist_write(newsd, buf, sizeof(buf)) != sizeof(buf)) {
                perror("server write to client");
                exit(1);
            }
        }
        else if (FD_ISSET(newsd, &rfds)) {
            memset(buf, '\0', sizeof(buf));
            n = read(newsd, buf, sizeof(buf));

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

            if (n == 0) {
                fprintf(stderr, "Client went away.\n");
                break;
            }

            if (decrypt_data(cfd)) {
                perror("server decrypt");
                exit(1);
            }

            if (insist_write(1, buf, n) != n) {
                perror("server write to server");
                exit(1);
            }
        }
    }

if (ioctl(cfd, CIOCFSESSION, &sess.ses)) {
    perror("server ioctl(CIOCFSESSION)");
    exit(1);
}
```

```
        if (close(cfd) < 0) {
            perror("server close(cfd)");
            exit(1);
        }

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

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

socket-client.c

Δημιουργούμε ένα TCP/IP socket και χρησιμοποιώντας την gethostbyname βρίσκουμε την διεύθυνση IP του server στον οποίο θέλουμε να συνδεθούμε από το το όνομα DNS που δόθηκε ως πρώτο όρισμα κατά την εκτέλεση του client. Στην συνέχεια, χρησιμοποιώντας την connect στέλνουμε αίτημα σύνδεσης στην διεύθυνση IP που πήραμε από την gethostbyname και στο port που δόθηκε ως δεύτερο όρισμα κατά την εκτέλεση του client. Αφού κατοχυρωθεί η επικοινωνία, ανοίγουμε την συσκευή dev/crypto και αρχικοποιούμε το struct session_op sess κατάλληλα ώστε να δημιουργήσουμε νέα συνεδρία (session) με την συσκευή καλώντας την ioctl με όρισμα CIOCGSESSION. Τέλος, μπαίνουμε σε αέναο βρόχο ανταλλαγής μηνυμάτων ο οποίος είναι συμμετρικός με αυτόν του server. Ακολουθεί ο κώδικας της συνάρτησης main του αρχείου socket-client.c:

```
int main(int argc, char *argv[])
{
    int sd, port, retval, cfd;
    ssize_t n;
    char *hostname, *filename;
    fd_set rfds;
    struct hostent *hp;
    struct sockaddr_in sa;

    if (argc > 4 || argc < 3) {
        fprintf(stderr, "Incorrect arguements");
        exit(1);
    }
    hostname = argv[1];
```

```
if( !(port = atoi(argv[2]))) {
    perror("atoi");
    exit(1);
}

/* 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");

filename = (argv[3] == NULL) ? "/dev/crypto" : argv[3];
cfd = open(filename, O_RDWR);
if (cfd < 0) {
    perror("client open(/dev/crypto)");
    exit(1);
}

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

if (ioctl(cfd, CIOCGSESSION, &sess)) {
```

```
        perror("client ioctl(CIOCGSESSION)");
        exit(1);
    }

    for (;;) {
        FD_ZERO(&rfds);
        FD_SET(0, &rfds);
        FD_SET(sd, &rfds);
        retval = select(sd+1, &rfds, NULL, NULL, NULL);
        if (retval < 0) {
            perror("client select");
            exit(1);
        }
        else if (FD_ISSET(0, &rfds)) {
            memset(buf, '\0', sizeof(buf));
            n = read(0, buf, sizeof(buf));

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

            if (n == 0) break;

            if (encrypt_data(cfd)) {
                perror("client encrypt");
                exit(1);
            }

            if (insist_write(sd, buf, sizeof(buf)) != sizeof(buf)) {
                perror("client write to server");
                exit(1);
            }
        }
        else if (FD_ISSET(sd, &rfds)) {
            memset(buf, '\0', sizeof(buf));
            n = read(sd, buf, sizeof(buf));

            if (n < 0) {
                perror("client read from server");
                exit(1);
            }
```

```
            if (n == 0) {
                fprintf(stderr, "Server went away.\n");
                break;
            }

            if (decrypt_data(cfd)) {
                perror("client decrypt");
                exit(1);
            }

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

    if (ioctl(cfd, CIOCFSESSION, &sess.ses)) {
        perror("client ioctl(CIOCFSESSION)");
        exit(1);
    }

    if (close(cfd) < 0) {
        perror("client close(cfd)");
        exit(1);
    }

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

Ζητούμενο 3

Backend

Για την υλοποίηση του backend μέρος του οδηγού προσθέσαμε κώδικα στα αρχεία hw/char/virtio-cryptodev.c και include/hw/virtio/virtio-cryptodev.h.

include/hw/virtio/virtio-cryptodev.h

Στο virtio-cryptodev.h προσθέτουμε την δήλωση σταθερών για των διαχωρισμό των επιμέρους ειδών κλήσεων της ioctl. Ακολουθεί ο κώδικας:

```
#ifndef VIRTIO_CRYPTODEV_H
#define VIRTIO_CRYPTODEV_H

#define DEBUG(str) \
   printf("[VIRTIO-CRYPTODEV] FILE[%s] LINE[%d] FUNC[%s] STR[%s]\n", \
          __FILE__, __LINE__, __func__, str);
#define DEBUG_IN() DEBUG("IN")

#define VIRTIO_CRYPTODEV_SYSCALL_TYPE_OPEN  0
#define VIRTIO_CRYPTODEV_SYSCALL_TYPE_CLOSE 1
#define VIRTIO_CRYPTODEV_SYSCALL_TYPE_IOCTL 2

#define VIRTIO_CRYPTODEV_IOCTL_CIOCGSESSION 3
#define VIRTIO_CRYPTODEV_IOCTL_CIOCFSESSION 4
#define VIRTIO_CRYPTODEV_IOCTL_CIOCCRYPT 5

#define TYPE_VIRTIO_CRYPTODEV "virtio-cryptodev"

#define CRYPTODEV_FILENAME  "/dev/crypto"

typedef struct VirtCryptodev {
   VirtIODevice parent_obj;
} VirtCryptodev;

#endif /* VIRTIO_CRYPTODEV_H */
```

hw/char/virtio-cryptodev.c

Στο virtio-cryptodev.c προσθέτουμε κώδικα στην συνάρτηση vq_handle_output. Ακολουθεί ο κώδικας

```
static void vq_handle_output(VirtIODevice *vdev, VirtQueue *vq)
{
    VirtQueueElement *elem;
    unsigned int *syscall_type, *cmd_type;
    unsigned char *key, *inv, *src, *dst;
    int *cfd, *host_ret;
    uint32_t *ses;
    struct session_op *sess;
    struct crypt_op *cryp;

    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");
    cfd = elem->in_sg[0].iov_base;
    *cfd = open("/dev/crypto", O_RDWR);
    if (*cfd < 0) {
        DEBUG("open error");
    }
    else {
        DEBUG("opened /dev/crypto");
    }
        break;

    case VIRTIO_CRYPTODEV_SYSCALL_TYPE_CLOSE:
        DEBUG("VIRTIO_CRYPTODEV_SYSCALL_TYPE_CLOSE");
        cfd = elem->out_sg[1].iov_base;
    if (close(*cfd) < 0) {
        DEBUG("close error");
```

```
}
else {
    DEBUG("closed /dev/crypto");
}
    break;

case VIRTIO_CRYPTODEV_SYSCALL_TYPE_IOCTL:
    DEBUG("VIRTIO_CRYPTODEV_SYSCALL_TYPE_IOCTL");
cfd = elem->out_sg[1].iov_base;
cmd_type = elem->out_sg[2].iov_base;
    switch (*cmd_type) {
case VIRTIO_CRYPTODEV_IOCTL_CIOCGSESSION:
    DEBUG("CIOCGSESSION");
    sess = elem->out_sg[3].iov_base;
    key = elem->out_sg[4].iov_base;
    sess = elem->in_sg[0].iov_base;
    host_ret = elem->in_sg[1].iov_base;

    sess->key = key;
    *host_ret = ioctl(*cfd, CIOCGSESSION, sess);
    if (*host_ret) {
    DEBUG("ioctl CIOCGSESSION error");
    }
    else {
    DEBUG("opened crypto session");
    }
    break;

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

        *host_ret = ioctl(*cfd, CIOCFSESSION, ses);
    if (*host_ret) {
            DEBUG("ioctl CIOCFSESSION error");
        }
    else {
            DEBUG("closed crypto session");
        }
        break;

case VIRTIO_CRYPTODEV_IOCTL_CIOCCRYPT:
```

```
            DEBUG("CIOCCRYPT");
            cryp = elem->out_sg[3].iov_base;
            inv = elem->out_sg[4].iov_base;
            src = elem->out_sg[5].iov_base;
            dst = elem->in_sg[0].iov_base;
            host_ret = elem->in_sg[1].iov_base;

            cryp->iv = inv;
            cryp->src = src;
            cryp->dst = dst;
            *host_ret = ioctl(*cfd, CIOCCRYPT, cryp);
            if (*host_ret) {
                    DEBUG("ioctl CIOCCRYPT error");
                }
            else {
                    DEBUG("encrypted/decrypted data");
                }
                break;

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

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

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

Αρχικά, λαμβάνουμε δεδομένα από την VirtQueue με την virtqueue_pop και εξάγουμε από το VirtQueueElement elem το πρώτο readable scatter gather list το οποίο περιέχει τον τύπο του system call που έχει κάνει ο guest. Στην συνέχεια, μέσω switch statement, ανάλογα με το system call εξυπηρετούμε το αίτημα του guest.

VIRTIO_CRYPTODEV_SYSCALL_TYPE_OPEN
Προσθέτουμε τον file descriptor (cfd) της συσκευής στην πρώτη writable scatter gather list του VirtQueue (cfd = elem->in_sg[0].iov_base;) ώστε να τον επιστρέψουμε στον guest και ανοίγουμε την συσκευή /dev/crypto.

VIRTIO_CRYPTODEV_SYSCALL_TYPE_CLOSE
Λαμβάνουμε τον file descriptor (cfd) της συσκευής που θέλει να κλείσει ο guest εξάγοντας από το VirtQueueElement elem το δεύτερο readable scatter gather list (cfd = elem->out_sg[1].iov_base;) και κλείνουμε την συσκευή /dev/crypto που αντιστοιχεί στον file descriptor.

VIRTIO_CRYPTODEV_SYSCALL_TYPE_IOCTL
Η εντολή ioctl μπορεί να κληθεί μπορεί να κληθεί με τρία διαφορετικά ορίσματα. Λαμβάνουμε τον file descriptor (cfd) της συσκευής που θέλει να κλείσει ο guest εξάγοντας από το VirtQueueElement elem το δεύτερο readable scatter gather list (cfd = elem->out_sg[1].iov_base;) και το όρισμα με το οποίο έγινε η κλήση ioctl εξάγοντας από το VirtQueueElement elem το τρίτο readable scatter gather list (cmd_type = elem->out_sg[2].iov_base;). Στην συνέχεια, μέσω switch statement, ανάλογα με το όρισμα (cmd_type) με το οποίο κλήθηκε η ioctl εξυπηρετούμε το αίτημα του guest.
VIRTIO_CRYPTODEV_IOCTL_CIOCGSESSION
Εάν έγινε κλήση με όρισμα CIOCGSESSION, λαμβάνουμε την δομή struct session_op (sess) και το κλειδί αποκρυπτογράφησης (key) εξάγοντας από το VirtQueueElement elem το τέταρτο και το πέμπτο readable scatter gather list αντίστοιχα (sess = elem->out_sg[3].iov_base; key = elem->out_sg[4].iov_base;). Προσθέτουμε την δομή struct session_op (sess) και την τιμή επιστροφής (host_ret) στην πρώτη και την δεύτερη writable scatter gather list του VirtQueue αντίστοιχα (sess = elem->in_sg[0].iov_base; host_ret = elem->in_sg[1].iov_base;) ώστε να επιστρέψουμε στον guest την δομή αφού αρχίσουμε το session με την συσκευή και την τιμή επιστροφής της ioctl που τρέχει ο host. Τέλος, περνάμε το κλειδί κρυπτογράφησης (key) στην δομή struct session_op (sess) και τρέχουμε την εντολή ioctl με τα ορίσματα που ζήτησε ο guest για να αρχίσουμε ένα session με την συσκευή.
VIRTIO_CRYPTODEV_IOCTL_CIOCFSESSION
Εάν έγινε κλήση με όρισμα CIOCFSESSION, λαμβάνουμε το αναγνωριστικό του session εξάγοντας από το VirtQueueElement elem το τέταρτο readable scatter gather list (ses = elem->out_sg[3].iov_base;) και προσθέτουμε την τιμή επιστροφής της εντολής ioctl που τρέχει ο host (host_ret) στην πρώτη writable scatter gather list του VirtQueue (host_ret = elem->in_sg[1].iov_base;) ώστε να την επιστρέψουμε στον guest. Τέλος, τρέχουμε την εντολή ioctl με τα ορίσματα που ζήτησε ο guest για να τερματίσουμε το session.

VIRTIO_CRYPTODEV_IOCTL_CIOCCRYPT
Εάν έγινε κλήση με όρισμα CIOCCRYPT, λαμβάνουμε την δομή struct crypt_op (cryp) που παίρνει ως όρισμα η ioctl, το initialization vector (inv) και το μήνυμα το οποίο στέλνει ο guest για κρυπτογράφηση/αποκρυπτογράφηση (src) εξάγοντας από το VirtQueueElement elem το τέταρτο, πέμπτο και το έκτο readable scatter gather list αντίστοιχα (cryp = elem->out_sg[3].iov_base; inv = elem->out_sg[4].iov_base; src = elem->out_sg[5].iov_base;). Προσθέτουμε το μήνυμα το οποίο έχει κρυπτογραφήσει/αποκρυπτογραφήσει (dst) και την τιμή επιστροφής (host_ret) στην πρώτη και την δεύτερη writable scatter gather list του VirtQueue αντίστοιχα (dst = elem->in_sg[0].iov_base; host_ret = elem->in_sg[1].iov_base;) ώστε να επιστρέψουμε στον guest το κρυπτογραφημένο/αποκρυπτογραφημένο μήνυμα και την τιμή επιστροφής της ioctl που τρέχει ο host. Τέλος, περνάμε initialization vector (inv), το μήνυμα που έστειλε ο guest (src) και το μήνυμα που στέλνει ο host (dst) στην δομή struct crypt_op (cryp) και τρέχουμε την εντολή ioctl με τα ορίσματα που ζήτησε ο guest για να κρυπτογραφήσουμε/αποκρυπτογραφήσουμε τα δεδομένα.

Αφού τρέξουμε το system call που έχει κάνει ο guest στέλνουμε δεδομένα στην VirtQueue με την virtqueue_push και ειδοποιούμε τον guest με την virtio_notify ο οποίος κάνει busy-wait για το αποτέλεσμα.

Frontend
Για την υλοποίηση του frontend μέρος του οδηγού προσθέσαμε κώδικα στα αρχεία crypto.h, crypto-module.c και crypto-chrdev.c.

crypto.h
Προσθέτουμε στο struct crypto_device σεμαφόρο τον οποίο χρησιμοποιούμε για τον συγχρονισμό της ανταλλαγής δεδομένων μεταξύ του host και του guest. Ο λόγος που χρησιμοποιούμε σεμαφόρο είναι διότι η ανταλλαγή γίνεται σε process context, οπότε θέλουμε οι διεργασίες να έχουν την δυνατότητα να κοιμηθούν για την πιο αποδοτική διαχείριση του χρόνου της CPU.

crypto-module.c
Στο crypto-module.c προσθέτουμε κώδικα στην συνάρτηση virtcons_probe. Αυτή η συνάρτηση καλείται όταν το kernel βρίσκει μία συσκευή με την οποία σχετιζόμαστε. Αρχικοποιεί το struct crypto_device και συνεπώς εκεί αρχικοποιούμε και τον σεμαφόρο του.

crypto-chrdev.c
Στο crypto-chrdev.c προσθέτουμε κώδικα στις συναρτήσεις crypto_chrdev_open, crypto_chrdev_release και crypto_chrdev_ioctl.

➔ crypto_chrdev_open

Μέσω της συνάρτησης crypto_chrdev_open, σχετίζεται το αρχείο που θα ανοίξει ο host με την σχετική συσκευή crypto. Αρχικά, περνάμε τις πληροφορίες της εικονικής συσκευής που ανοίγουμε στο struct crypto_device *crdev το οποίο περιέχει το VirtQueue της συσκευής, τον σεμαφόρο της, πληροφορίες για την συσκευή κτλ. Στην συνέχεια, για να συσχετίσουμε το αρχείο με την συσκευή, περνάμε το struct crdev στο struct crypto_open_file *crof το οποίο συσχετίζει την συσκευή και τις πληροφορίες της με τον file descriptor της συσκευής που θα ανοίξει ο host και θα στείλει στον guest. Αρχικοποιούμε μία readable scatter gather list και μία writable scatter gather list (sg_init_one(&syscall_type_sg, syscall_type και sizeof(syscall_type)), sg_init_one(&host_fd_sg, &crof->host_fd, sizeof(crof->host_fd)) αντίστοιχα) τις οποίες κάνουμε bind στις μεταβλητές που αντιστοιχούν στον τύπο του system call (VIRTIO_CRYPTODEV_SYSCALL_OPEN) και τον file descriptor του αρχείου που θέλουμε να μας επιστρέψει ο host (host_fd). Αφού δημιουργήσουμε τις sg lists κλειδώνουμε τον σεμαφόρο της συσκευής (για αποφυγή race condition με διεργασία παιδί κατά την ανταλλαγή πληροφοριών με τον host), προσθέτουμε τα sg list στο VirtQueue της συσκευής (virtqueue_add_sgs(crdev->vq, sgs, 1, 1, &syscall_type_sg, GFP_ATOMIC)), ενημερώνουμε τον host για την αποστολή δεδομένων (virtqueue_kick(crdev->vq) και περιμένουμε σε busy-wait-loop την απάντηση του (while (virtqueue_get_buf(crdev->vq, &len) == NULL);). Τέλος, ελέγχουμε εάν ο file descriptor που μας έστειλε ο host είναι έγκυρος (crof->host_fd < 0), δηλαδή ελέγχουμε για σφάλμα του syscall open από την πλευρά του host. Ακολουθεί ο κώδικας:

```
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;
    struct scatterlist syscall_type_sg, host_fd_sg, *sgs[2];
    unsigned int *syscall_type;
    int *host_fd;

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

    /* Wait for the host to process our data. Lock semaphore to prevent
race conditions with father-child proccesses */
    if (down_interruptible(&crdev->lock)) {
        ret = -EINTR;
        debug("down_interruptible() interrupted by signal");
        goto fail;
    }

    err = virtqueue_add_sgs(crdev->vq, sgs, 1, 1, &syscall_type_sg,
GFP_ATOMIC);
    if (err < 0) {
        ret = err;
                debug("virtqueue_add_sgs() error");
        up(&crdev->lock);
                goto fail;
    }
    if(!virtqueue_kick(crdev->vq)) {
```

```
        ret = -EAGAIN;
                debug("virtqueue_kick() error");
        up(&crdev->lock);
                goto fail;
    }
    /* Do nothing until we receive response from host */
    while (virtqueue_get_buf(crdev->vq, &len) == NULL);

    up(&crdev->lock);

    /* If host failed to open() return -ENODEV. */
    if (crof->host_fd < 0) {
        ret = -ENODEV;
        debug("host open error %d", crof->host_fd);
    }
    else {
        debug("host open /dev/crypto with cfd %d", crof->host_fd);
    }

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

➔ crypto_chrdev_release

Μέσω της συνάρτησης crypto_chrdev_release, κλείνουμε (ζητάμε από τον host να κλείσει) την συσκευή που έχει ανοίξει ο host και απελευθερώνουμε τον χώρο που έχουμε κατανέμει (με την crypto_chrdev_open) για το struct crypto_open_file *crof. Αρχικοποιούμε δύο readable scatter gather lists (sg_init_one(&syscall_type_sg, syscall_type, sizeof(syscall_type)) και sg_init_one(&host_fd_sg, &crof->host_fd, sizeof(crof->host_fd))) τις οποίες κάνουμε bind στις μεταβλητές που αντιστοιχούν στον τύπο του system call (VIRTIO_CRYPTODEV_SYSCALL_CLOSE) και τον file descriptor του αρχείου που θέλουμε να κλείσει ο host (host_fd). Αφού δημιουργήσουμε τις sg lists κλειδώνουμε τον σεμαφόρο της συσκευής (για αποφυγή race condition με διεργασία παιδί κατά την ανταλλαγή πληροφοριών με τον host), προσθέτουμε τα sg list στο VirtQueue της συσκευής (virtqueue_add_sgs(crdev->vq, sgs, 2, 0, &syscall_type_sg, GFP_ATOMIC)), ενημερώνουμε τον host για την αποστολή δεδομένων (virtqueue_kick(crdev->vq) και περιμένουμε σε busy-wait-loop την απάντηση του (while (virtqueue_get_buf(crdev->vq, &len) == NULL);). Τέλος, απελευθερώνουμε τον χώρο που έχουμε κατανέμει στο struct crof. Ακολουθεί ο κώδικας:

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

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

   /* Wait for the host to process our data. */
   if (down_interruptible(&crdev->lock)) {
       ret = -EINTR;
       debug("down_interruptible() interrupted by signal");
               goto fail_close;
   }

   err = virtqueue_add_sgs(crdev->vq, sgs, 2, 0, &syscall_type_sg,
GFP_ATOMIC);
   if (err < 0) {
               ret = err;
               debug("virtqueue_add_sgs() error");
       up(&crdev->lock);
               goto fail_close;
       }
   if(!virtqueue_kick(crdev->vq)) {
               ret = -EAGAIN;
               debug("virtqueue_kick() error");
               up(&crdev->lock);
               goto fail_close;
       }
       /* Do nothing until we receive response from host */
       while (virtqueue_get_buf(crdev->vq, &len) == NULL);

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

```
fail_close:
      kfree(crof);
 debug("Leaving");
       return ret;
}
```

➔ crypto_chrdev_ioctl

Μέσω της συνάρτησης crypto_chrdev_ioctl, στέλνουμε εντολές στην συσκευή κρυπτογράφησης. Αρχικοποιούμε δύο readable scatter gather lists (sg_init_one(&syscall_type_sg, syscall_type, sizeof(syscall_type)) και sg_init_one(&host_fd_sg, &crof->host_fd, sizeof(crof->host_fd))) τις οποίες κάνουμε bind στις μεταβλητές που αντιστοιχούν στον τύπο του system call (VIRTIO_CRYPTODEV_SYSCALL_IOCTL) και τον file descriptor της συσκευής που έχει ανοίξει ο host (host_fd). Στην συνέχεια ξεχωρίζουμε ποιο είδος κλήσης ioctl έχει γίνει με ένα switch για το όρισμα cmd (εντολή) της ioctl.

CIOCGSESSION

Αρχικοποιούμε ένα readable scatter gather list (sg_init_one(&cmd_type_sg, cmd_type, sizeof(*cmd_type))) την οποία κάνουμε bind στην μεταβλητή (cmd_type) που αντιστοιχεί στο όρισμα της ioctl (VIRTIO_CRYPTODEV_IOCTL_CIOCGSESSION). Το τελευταίο όρισμα της εντολής (arg) είναι η διεύθυνση στην οποία βρίσκεται το struct session_op *sess . Αντιγράφουμε το struct από το χώρο χρήστη, και αντιγράφουμε και το κλειδί αποκρυπτογράφησης (key) από τον χώρο χρήστη ξεχωριστά διότι το sess δεν περιέχει το κλειδί αλλά δείκτη σε αυτό. Αρχικοποιούμε δύο readable scatter gather lists (sg_init_one(&sess_sg, sess, sizeof(*sess)), sg_init_one(&key_sg, key, sess->keylen)) τα οποία κάνουμε bind στο struct sess που στέλνουμε στον host και το κλειδί αποκρυπτογράφησης (key) και δύο writable scatter gather lists (sg_init_one(&sess_sg, sess, sizeof(*sess)), sg_init_one(&host_ret_sg, host_ret, sizeof(*host_ret))) τα οποία κάνουμε bind στο struct sess που λαμβάνουμε από τον host και την τιμή επιστροφής (host_ret) την οποία επιστρέφει η εντολή ioctl που τρέχει ο host εκ μέρους του guest.

CIOCFSESSION

Αρχικοποιούμε ένα readable scatter gather list (sg_init_one(&cmd_type_sg, cmd_type, sizeof(*cmd_type))) την οποία κάνουμε bind στην μεταβλητή (cmd_type) που αντιστοιχεί στο όρισμα της ioctl (VIRTIO_CRYPTODEV_IOCTL_CIOCFSESSION). Το τελευταίο όρισμα της εντολής (arg) είναι η διεύθυνση στην οποία βρίσκεται το αναγνωριστικό του session (ses) που έχει ανοίξει με την συσκευή crypto. Αντιγράφουμε το ses από το χώρο χρήστη και αρχικοποιούμε ένα readable scatter gather list (sg_init_one(&ses_sg, ses, sizeof(*ses))) το οποίο το κάνουμε bind στο ses και ένα writable scatter gather list (sg_init_one(&host_ret_sg, host_ret, sizeof(*host_ret))) το οποίο το κάνουμε bind στην τιμή επιστροφής (host_ret) την οποία επιστρέφει η εντολή ioctl που τρέχει ο host εκ μέρους του guest.

CIOCCRYPT

Αρχικοποιούμε ένα readable scatter gather list (sg_init_one(&cmd_type_sg, cmd_type, sizeof(*cmd_type))) την οποία κάνουμε bind στην μεταβλητή (cmd_type) που αντιστοιχεί στο όρισμα της ioctl (VIRTIO_CRYPTODEV_IOCTL_CIOCCRYPT). Το τελευταίο όρισμα της εντολής (arg) είναι η διεύθυνση στην οποία βρίσκεται το struct crypt_op *cryp. Αντιγράφουμε το struct από το χώρο χρήστη. Αντιγράφουμε και το μήνυμα που θέλουμε να κρυπτογραφήσουμε/αποκρυπτογραφήσουμε (src) και το initialisation vector (inv) από τον χώρο χρήστη ξεχωριστά διότι το cryp δεν περιέχει το μήνυμα και το initialisation vector αλλά δείκτες σε αυτά. Αρχικοποιούμε τρία readable scatter gather lists (sg_init_one(&cryp_sg, cryp, sizeof(*cryp)), sg_init_one(&inv_sg, inv, sizeof(*inv)), sg_init_one(&src_sg, src, cryp->len)) τα οποία κάνουμε bind στο struct cryp που στέλνουμε στον host, στο initialisation vector (inv) και στο μήνυμα που στέλνουμε στον host για να το κρυπτογραφήσει/αποκρυπτογραφήσει. Αρχικοποιούμε δύο writable scatter gather lists (sg_init_one(&dst_sg, dst, cryp->len), sg_init_one(&host_ret_sg, host_ret, sizeof(*host_ret))) τα οποία κάνουμε bind στο κρυπτογραφημένο/αποκρυπτογραφημένο μήνυμα που λαμβάνουμε από τον host και στην τιμή επιστροφής (host_ret) την οποία επιστρέφει η εντολή ioctl που τρέχει ο host εκ μέρους του guest. Στην συνέχεια, κλειδώνουμε τον σεμαφόρο της συσκευής (για αποφυγή race condition με διεργασία παιδί κατά την ανταλλαγή πληροφοριών με τον host), προσθέτουμε τα sg list στο VirtQueue της συσκευής (virtqueue_add_sgs(vq, sgs, num_out, num_in, &syscall_type_sg, GFP_ATOMIC)), ενημερώνουμε τον host για την αποστολή δεδομένων (virtqueue_kick(crdev->vq) και περιμένουμε σε busy-wait-loop την απάντηση του (while (virtqueue_get_buf(crdev->vq, &len) == NULL);). Τέλος, εάν έγινε κλήση με όρισμα CIOCGSESSION αντιγράφουμε στον χώρο χρήστη το struct sess που επιστρέφει ο host, εάν έγινε κλήση CIOCCRYPT αντιγράφουμε στον χώρο χρήστη το κρυπτογραφημένο/αποκρυπτογραφημένο μήνυμα και απελευθερώνουμε τον χώρο που έχουμε δεσμεύσει για τις μεταβλητές και τα struct που χρειαζόμαστε για τις εντολές. Ακολουθεί ο κώδικας:

```
static long crypto_chrdev_ioctl(struct file *filp, unsigned int cmd,
unsigned long arg)
{
    long ret = 0;
    int err, *host_ret = NULL;
    struct crypto_open_file *crof = filp->private_data;
    struct crypto_device *crdev = crof->crdev;
    struct virtqueue *vq = crdev->vq;
    struct scatterlist syscall_type_sg, cmd_type_sg,  host_fd_sg, sess_sg,
key_sg, ses_sg, cryp_sg, inv_sg, host_ret_sg, src_sg, dst_sg, *sgs[11];
    unsigned int num_out, num_in, len;
    unsigned char *src = NULL, *dst = NULL, *key = NULL, *inv = NULL;
    unsigned int *syscall_type, *cmd_type = NULL;
    uint32_t *ses = NULL;
    struct session_op *sess = NULL;
```

```
    struct crypt_op *cryp = NULL;
    #define INV_SIZE 16

    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;

    num_out = 0;
    num_in = 0;

    /* 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");

        cmd_type = kzalloc(sizeof(*cmd_type), GFP_KERNEL);
        *cmd_type = VIRTIO_CRYPTODEV_IOCTL_CIOCGSESSION;
        sg_init_one(&cmd_type_sg, cmd_type, sizeof(*cmd_type));
        sgs[num_out++] = &cmd_type_sg;

        sess = kzalloc(sizeof(*sess), GFP_KERNEL);
        if (copy_from_user(sess, (struct session_op *)arg, sizeof(*sess)))
{
                    debug("copy_from_user sess failed");
                    ret = -EFAULT;
                    goto fail_ioctl;
            }
        sg_init_one(&sess_sg, sess, sizeof(*sess));
                sgs[num_out++] = &sess_sg;

        key = kzalloc(sess->keylen, GFP_KERNEL);
        if (copy_from_user(key, sess->key, sess->keylen)) {
                        debug("copy_from_user key failed");
                        ret = -EFAULT;
                        goto fail_ioctl;
```

```
            }
        sg_init_one(&key_sg, key, sess->keylen);
            sgs[num_out++] = &key_sg;

        sg_init_one(&sess_sg, sess, sizeof(*sess));
            sgs[num_out + num_in++] = &sess_sg;

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

        break;

    case CIOCFSESSION:
        debug("CIOCFSESSION");

        cmd_type = kzalloc(sizeof(*cmd_type), GFP_KERNEL);
            *cmd_type = VIRTIO_CRYPTODEV_IOCTL_CIOCFSESSION;
            sg_init_one(&cmd_type_sg, cmd_type, sizeof(*cmd_type));
            sgs[num_out++] = &cmd_type_sg;

        ses = kzalloc(sizeof(uint32_t), GFP_KERNEL);
        if (copy_from_user(ses, (uint32_t *)arg, sizeof(*ses))) {
                debug("copy_from_user ses failed");
                ret = -EFAULT;
                goto fail_ioctl;
            }
        sg_init_one(&ses_sg, ses, sizeof(*ses));
            sgs[num_out++] = &ses_sg;

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

        break;

    case CIOCCRYPT:
        debug("CIOCCRYPT");

        cmd_type = kzalloc(sizeof(*cmd_type), GFP_KERNEL);
            *cmd_type = VIRTIO_CRYPTODEV_IOCTL_CIOCCRYPT;
            sg_init_one(&cmd_type_sg, cmd_type, sizeof(*cmd_type));
            sgs[num_out++] = &cmd_type_sg;
```

```
    cryp = kzalloc(sizeof(*cryp), GFP_KERNEL);
    if (copy_from_user(cryp, (struct crypt_op *)arg, sizeof(*cryp))) {
                debug("copy_from_user cryp failed");
                ret = -EFAULT;
                goto fail_ioctl;
        }
    sg_init_one(&cryp_sg, cryp, sizeof(*cryp));
            sgs[num_out++] = &cryp_sg;

    inv = kzalloc(INV_SIZE, GFP_KERNEL);
            if (copy_from_user(inv, cryp->iv, INV_SIZE)) {
                    debug("copy_from_user inv failed");
                    ret = -EFAULT;
                    goto fail_ioctl;
            }
    sg_init_one(&inv_sg, inv, sizeof(*inv));
            sgs[num_out++] = &inv_sg;

    src = kzalloc(cryp->len, GFP_KERNEL);
            if (copy_from_user(src, cryp->src, cryp->len)) {
                    debug("copy_from_user src failed");
                    ret = -EFAULT;
                    goto fail_ioctl;
            }
    sg_init_one(&src_sg, src, cryp->len);
            sgs[num_out++] = &src_sg;

    dst = kzalloc(cryp->len, GFP_KERNEL);
    sg_init_one(&dst_sg, dst, cryp->len);
            sgs[num_out + num_in++] = &dst_sg;

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

    break;

default:
    debug("Unsupported ioctl command");

    break;
}
```

```
   /* Wait for the host to process our data. Lock semaphore to prevent
race conditions with father-child proccesses */
        if (down_interruptible(&crdev->lock)) {
                ret = -EINTR;
                debug("down_interruptible() interrupted by signal");
                goto fail_ioctl;
        }

   err = virtqueue_add_sgs(vq, sgs, num_out, num_in,
                           &syscall_type_sg, GFP_ATOMIC);
   if (err < 0) {
                ret = err;
                debug("virtqueue_add_sgs() error");
        up(&crdev->lock);
                goto fail_ioctl;
        }
   if(!virtqueue_kick(crdev->vq)) {
                ret = -EAGAIN;
                debug("virtqueue_kick() error");
                up(&crdev->lock);
                goto fail_ioctl;
        }
   /* Do nothing until we receive response from host */
   while (virtqueue_get_buf(vq, &len) == NULL) ;

   up(&crdev->lock);

   switch (cmd) {
   case CIOCGSESSION:
        if (copy_to_user((struct session_op *)arg, sess, sizeof(*sess)))
{
                    debug("copy to user sess error");
                    ret = -EFAULT;
                    goto fail_ioctl;
           }
       break;
   case CIOCCRYPT:
        if (copy_to_user(((struct crypt_op *)arg)->dst, dst, cryp->len))
{
                    debug("copy to user cryp error");
                    ret = -EFAULT;
                    goto fail_ioctl;
```

```
            }
        break;
    }
    ret = *host_ret;

fail_ioctl:
    kfree(host_ret);
    kfree(dst);
    kfree(src);
    kfree(inv);
    kfree(ses);
    kfree(key);
    kfree(sess);
    kfree(cmd_type);
    kfree(syscall_type);
    debug("Leaving");
    return ret;
}
```