ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Εργαστήριο Λειτουργικών Συστημάτων

Αναφορά για την 3η Άσκηση: Κρυπτογραφική συσκευή VirtIO για QEMU-KVM

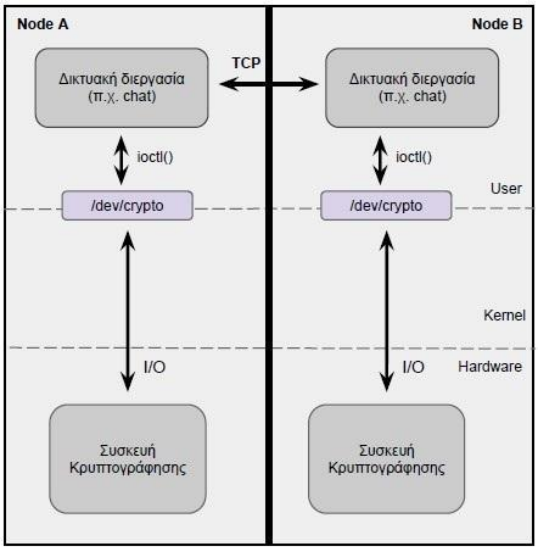
Ον/μο :Τελάλη Ειρήνη
Α.Μ.: :03113009

Ον/μο :Μπερέτσος Θεόδωρος
Α.Μ.: :03111612

Ομάδα: b27
Ημερομηνία εξέτασης: 22/6/2017
Ημερομηνία αποστολής αναφοράς: 27/6/2017

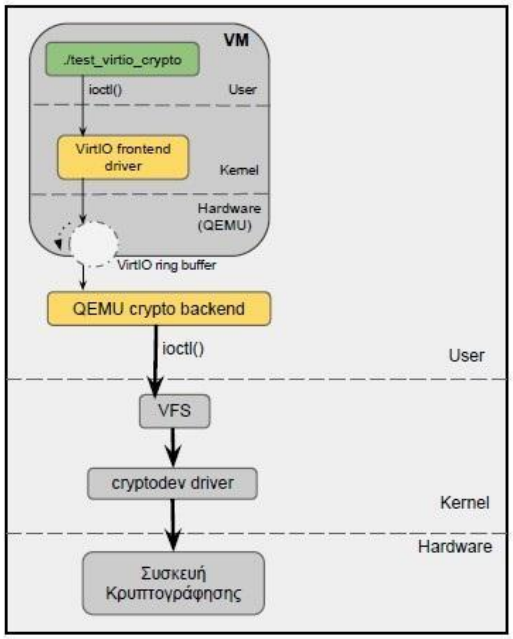
1. Σκοπός

Αντικείμενο του πρώτου μέρους της τρίτης εργαστηριακής άσκησης είναι η κατασκευή μιας εφαρμογής chat τα δύο άκρα της οποία επικοινωνούν μέσω TCP/IP, κάνοντας χρήση του BSD Sockets API. Επίσης γίνεται επέκταση της εφαρμογής αυτής με κρυπτογράφηση των μηνυμάτων που ανταλλάσσονται μέσω της συσκευής cryptodev (/dev/crypto) η οποία εισάγεται ως module στον πυρήνα του λειτουργικού.



Κρυπτογραφημένο chat πάνω από TCP/IP

Αντικείμενο του δεύτερου μέρους της τρίτης εργαστηριακής άσκησης είναι η υλοποίηση ενός οδηγού μιας εικονικής συσκευής (device driver) κρυπτογράφησης για εικονικές μηχανές που εκτελούνται από το QEMU σε περιβάλλον του λειτουργικού Linux. Η συσκευή αυτή θα εκτελεί ένα υποσύνολο των λειτουργιών που προσφέρει η συσκευή cryptodev που χρησιμοποιήσαμε στο πρώτο μέρος της άσκησης:



Αρχιτεκτονική λογισμικού της (paravirtualized) virtio-crypto συσκευής

Τα ζητούμενα της άσκησης περιλαμβάνουν την υποστήριξη μόνο ενός μέρους από

τις λειτουργίες της συσκευής /dev/crypto. Συγκεκριμένα, η υλοποίησή σας θα πρέπει να υποστηρίζει τέσσερις κλήσεις ioctl, (i) CIOCGSESSION για την αρχή ενός session, (ii) CIOCCRYPT για κρυπτογράφηση και αποκρυπτογράφηση και τέλος (iii) CIOCFSESSION για τον τερματισμό του session.

Από το σχήμα γίνεται προφανής ο διαχωρισμός του οδηγού σε δύο μέρη, backend και frontend. Το πρώτο μέρος εκτελείται στο χώρο χρήστη του host, ενώ το δεύτερο στο χώρο πυρήνα της εικονικής μηχανής. Τα δύο μέρη επικοινωνούν μεταξύ τους μέσω των VirtQueues.

Η τεχνική της παραεικονικοποίησης (paravirtualization) έγκειται στο ότι η συσκευή που θα υλοποιήσουμε γνωρίζει ότι τρέχει σε Virtual Machine (VM) και όχι σε κανονικό σύστημα.

2. Ζητούμενο 1 - Chat μέσω TCP / IP sockets

Για την κατασκευή του chat επιλέχθηκε το μοντέλο του εξυπηρετητή – πελάτη (server- client). Εν προκειμένω ο server δέχεται κλήσεις σύνδεσης (connections) από τους clients που τις πραγματοποιούν.

2.1. Πηγαίος κώδικας (source code)

2.1.1. socket-server.c

```
int main(void)
{
    int sockfd, newsockfd;

    char recv_buf[BUFSIZE], send_buf[BUFSIZE];
    char addrstr[INET_ADDRSTRLEN];

    ssize_t n;
    socklen_t len;
    struct sockaddr_in server_addr, client_addr;

    fd_set master;

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

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

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

    /* Listen for incoming connections */
    if (listen(sockfd, 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 ((newsockfd = accept(sockfd, (struct sockaddr *)&client_addr, &len))
< 0) {
            perror("accept");
            exit(1);
        }
        if (!inet_ntop(AF_INET, &client_addr.sin_addr, addrstr,
sizeof(addrstr))) {
            perror("could not format IP address");
            exit(1);
        }
        fprintf(stderr, "Incoming connection from %s:%d\n", addrstr,
ntohs(client_addr.sin_port));

        printf("\nNew Session started!!!\n");

        /* We break out of the loop when the remote peer goes away */
        for (;;) {
            FD_ZERO(&master);
            FD_SET(newsockfd, &master);
            FD_SET(0, &master);
            fflush(stdout);
            int readsockets = select(newsockfd + 1, &master, NULL, NULL, NULL);

            if (readsockets < 0) {
                perror("select");
                exit(1);
            }
            else {
                /* client input */
                if (FD_ISSET(newsockfd, &master)){
                    /* read from client */
                    n = read(newsockfd, recv_buf, sizeof(recv_buf));
                    if (n <= 0) {
                        if (n < 0)
                            perror("read from remote peer failed");
                        else
                            fprintf(stderr, "\nPeer went away...\n");
                        break;
                    }
                    fprintf(stdout, "\nRemote: ");
                    fflush(stdout);
                    /* write to stdout */
                    if (insist_write(STDOUT_FILENO, recv_buf, n) != n) {
                        perror("write to local buff failed");
                        break;
                    }

                }
                else{
                    /* server output */
                    n = read(0, send_buf, sizeof(send_buf));
                    /* read from stdin */
                    if (n <= 0) {
                        if (n < 0)
                            perror("read from localy  failed");
                        else
                            fprintf(stderr, "I went away\n");
                        break;
                    }
                    /* write to client */
```

```
                    if (insist_write(newsockfd, send_buf, n) != n) {
                        perror("write to remote failed");
                        break;
                    }
                    fflush(stdout);
                }
            }
        }
        if (close(newsockfd) < 0) {
            perror("close");
        }
    }
    /* This will never happen */
    return 1;
}
```

2.1.2. socket-client.c

```
int main(int argc, char *argv[])
{
    int sd, port;
    ssize_t n;
    char send_buf[BUFSIZE], recv_buf[BUFSIZE];
    char *hostname;
    struct hostent *hp;
    struct sockaddr_in sa;

    fd_set master;

    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");

    /* Read answer and write it to standard output */
    for (;;) {
        FD_ZERO(&master);
        FD_SET(sd, &master);
        FD_SET(0, &master);
```

```
        fflush(stdout);
        int readsockets = select(sd + 1, &master, NULL, NULL, NULL);

        if (readsockets < 0) {
            perror("select");
            exit(EXIT_FAILURE);
        }
        else{
            /* server output */
            if (FD_ISSET(sd, &master)){
            /* read from server*/
            n = read(sd, recv_buf, sizeof(recv_buf));
            if (n <= 0) {
                if (n < 0)
                    perror("read from remote peer failed");
                else
                    fprintf(stderr, "\nPeer went away...\n");
                    break;
                }
                fprintf(stdout, "\nRemote: ");
                fflush(stdout);
                /* write to stdout what server said */
                if (insist_write(STDOUT_FILENO, recv_buf, n) != n) {
                    perror("write to local buff failed");
                    break;
                }
            }
            else{
                /* client input from stdin */
                n = read(0, send_buf, sizeof(send_buf));
                if (n <= 0) {
                    if (n < 0)
                        perror("read from localy  failed");
                    else
                        fprintf(stderr, "I went away\n");
                    break;
                }
                /* write to client */
                if (insist_write(sd, send_buf, n) != n) {
                    perror("write to remote  failed");
                    break;
                }
                fflush(stdout);
            }
        }
    }
    /* Make sure we don't leak open files */
    if (close(sd) < 0)
        perror("close");

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

2.2. Περιγραφή υλοποίησης των επιμέρους τμημάτων του παραπάνω κώδικα

2.2.1. Server: Δημιουργία Socket

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

Η κλήσης της socket() πραγματοποιεί τη δημιουργία ενός socket επιστρέφοντας έναν file descriptor. Τα ορίσματα που παίρνει ορίζουν τον τύπο του socket που θα δημιουργήσουν και στην περίπτωσή μας είναι TCP / IPv4 κατάλληλο για stream δεδομένων όπως ενός chat.

2.2.2. Server: Port Binding και αναμονή για εισερχόμενες συνδέσεις

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

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

Η κλήση της `bind()` δεσμεύει μια θύρα του συστήματος ώστε να χρησιμοποιηθεί από το socket. Σε αυτή τη θύρα θα συνδεθεί ο client.

Η κλήση της `listen()` κάνει το socket που έχουμε δημιουργήσει να περιμένει εισερχόμενες συνδέσεις, οι οποίες μπαίνουν σε μια ουρά (`TCP_BACKLOG == 5`) περιμένοντας να γίνουν accept.

2.2.3. Server: Αποδοχή σύνδεσης

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

        /* Accept an incoming connection */
        len = sizeof(struct sockaddr_in);
        if ((newsockfd = accept(sockfd, (struct sockaddr *)&client_addr,
&len)) < 0) {
            perror("accept");
            exit(1);
        }
        if (!inet_ntop(AF_INET, &client_addr.sin_addr, addrstr,
sizeof(addrstr))) {
            perror("could not format IP address");
            exit(1);
        }
        fprintf(stderr, "Incoming connection from %s:%d\n", addrstr,
ntohs(client_addr.sin_port));
```

Η κλήσης της `accept()` επιστρέφει ένα νέο socket, δηλαδή ένα νέο file descriptor το οποίο θα χρησιμοποιήσουμε για να στείλουμε και να λάβουμε μηνύματα προς και από τον client.

2.2.4. Server: Κλείσιμο Σύνδεσης

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

Η κλήση της `close()` απελευθερώνει το file descriptor για το socket που δημιούργησε η `accept()` μετά το τέλος μιας συνομιλίας. Μετά την `close()` ο server

μπορεί ακόμα να δεχθεί νέες συνδέσεις αφού η εκτέλεση του προγράμματος βρίσκεται εντός της ίδιας for loop.

2.2.5. Client: Δημιουργία Socket

```
int sd;
/* 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");
```

Ομοίως με τον server και εδώ η socket() δημιουργεί έναν file descriptor τον οποίο θα χρησιμοποιήσουμε για να συνδεθούμε στον server.

2.2.6. Client: Αναζήτηση του ονόματος του server μέσω DNS

```
/* Look up remote hostname on DNS */
if ( !(hp = gethostbyname(hostname))) {
    printf("DNS lookup failed for host %s\n", hostname);
    exit(1);
}
```

Η `gethostbyname()` μετατρέπει το όνομα του εξυπηρετητή που δώσαμε ως όρισμα σε διεύθυνση IPv4. Στην περίπτωση μας που η διεύθυνση δίνεται από εμάς κατευθείαν σε αυτή τη μορφή, περνιέται κατευθείαν στο κατάλληλο struct.

2.2.7. Client: Σύνδεση στον server

```
/* 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");
```

Η κλήση της `connect()` πραγματοποιεί τη σύνδεση στον server και πλέον μπορούμε να επικοινωνούμε με αυτόν μέσω του ίδιου file descriptor `sd`. Στο `struct sockaddr` είναι αποθηκευμένη η διεύθυνση του server. Εκεί είναι αποθηκευμένο και το πρωτόκολλο επικοινωνίας το οποίο θα καθορίσει τον τύπο της σύνδεσης.

2.2.8. Server & Client: Select from FDs

```
FD_ZERO(&master);
FD_SET(sd, &master);
FD_SET(0, &master);
int readsockets = select(sd + 1, &master, NULL, NULL, NULL);

if (readsockets < 0) {
    perror("select");
    exit(EXIT_FAILURE);
}
else{
    if (FD_ISSET(sd, &master)){
/* ... */
 }
```

Η `select()` υλοποιεί έναν μηχανισμό με τον οποίο γίνεται ενημέρωση όταν κάποιος από τους συνομιλητές είναι έτοιμος να εκτελέσει κάποια ενέργεια. Η `select()`

παραμένει μπλοκαρισμένη μέχρι κάποιος να είναι έτοιμος. Αν επιστραφεί αρνητική τιμή έχουμε σφάλμα. Αλλιώς, χρησιμοποιούμε τον FD_ISSET με την οποία μαθαίνουμε ποιος είναι έτοιμος και εκτελούμε την λειτουργία που ζητείται.

2.2.9. Server & Client: Λήψη και εμφάνιση μηνύματος

```
/* read from remote*/
n = read(sd, recv_buf, sizeof(recv_buf));
if (n <= 0) {
    if (n < 0)
        perror("read from remote peer failed");
    else
        fprintf(stderr, "\nPeer went away...\n");
        break;
    }
    fprintf(stdout, "\nRemote: ");
    fflush(stdout);
    /* write to stdout what server said */
    if (insist_write(STDOUT_FILENO, recv_buf, n) != n) {
        perror("write to local buff failed");
        break;
    }
}
```

Μόλις υπάρχει διαθέσιμο μήνυμα προς ανάγνωση στο socket της σύνδεσης η `read()` θα το παραλάβει και θα τα μεταφέρει σε έναν buffer. Από εκεί θα τα εκτυπώσουμε μέσω της `insist_write()` (που ήταν ήδη υλοποιημένη από το βοηθητικό κώδικα της άσκησης) στο `stdout` και επομένως στο terminal του παραλήπτη. Όταν επιστρέφεται 0 σημαίνει ότι ο συνομιλητής έχει εγκαταλείψει τη συνομιλία.

2.2.10. Server & Client: Πληκτρολόγηση και αποστολή μηνύματος

```
/* read input from stdin */
n = read(0, send_buf, sizeof(send_buf));
if (n <= 0) {
    if (n < 0)
        perror("read from localy  failed");
    else
        fprintf(stderr, "I went away\n");
    break;
}
/* write to client */
if (insist_write(sd, send_buf, n) != n) {
    perror("write to remote  failed");
    break;
}
```

Μόλις υπάρχει διαθέσιμο μήνυμα προς αποστολή, δηλαδή μόλις πατηθεί enter μετά την πληκτρολόγησή του, το διαβάζουμε από το stdin μέσω της read() και το τοποθετούμε σε έναν buffer. Από εκεί θα τα στείλουμε μέσω της insist_write() στον απομακρυσμένο χρήστη γράφοντας στον αντίστοιχο file descriptor sd. Εάν ο χρήστης έχει πληκτρολογήσει περισσότερα δεδομένα από sizeof(send_buf), στις επόμενες επαναλήψεις θα επαναλαμβάνονται τα read και write μέχρι να αποσταλούν όλα τα δεδομένα.

2.2.11. Server & Client: Σενάριο εκτέλεσης του chat που υλοποιήσαμε

Έστω ότι ξεκινάμε τον server. Τότε, θα δημιουργηθεί ένα νέο socket το οποίο θα χρησιμοποιηθεί από το server ως το βασικό κανάλι του chat και θα ‘δεθεί’ (`bind()`) με μία port η οποία είναι γνωστή και στην οποία θα ζητήσει αργότερα ο client σύνδεση.

Έπειτα ο server περιμένει με τη `listen()` έως ότου έρθει μια αίτηση για σύνδεση. Έστω, τώρα, ότι ένας client ξεκινάει, παίρνει ως ορίσματα τη διεύθυνση του server και την port και ζητάει να συνδεθεί στο server. Ο server θα δημιουργήσει ένα νέο socket για την επικοινωνία του με το συγκεκριμένο client και θα περιμένει να διαβάσει κάτι που θα του στείλει ο client χρησιμοποιώντας την `read()`. Αν διαβάσει τότε με την `insist_write()` τυπώνει αυτό που διάβασε στο stdout. Αν πρόκειται να στείλει δεδομένα ο server τότε με την read() διαβάζει από το stdin και χρησιμοποιώντας το `send_buf` στέλνει αυτό που γράφηκε στον client με την `insist_write()`. Αντίστοιχη είναι και η λειτουργικότητα του client ως προς τη λήψη και την αποστολή μηνυμάτων.

Ας υποθέσουμε ότι ένας δεύτερος client2 ζητάει να συνδεθεί με το server. Τότε, ο server, όπως έχει υλοποιηθεί, θα δημιουργήσει ένα **νέο socket** για την επικοινωνία του με το νέο client2, και θα κάνει `accept()` τη σύνδεση αφού κλείσει ο client1 της σύνδεσή του. Στη συνέχεια κάθε φορά που ο client2 θα στέλνει κάτι ο server θα το δέχεται μέσω της `select()` και θα αναγνωρίζει τον αποστολέα μέσω της FD_ISSET.

3. Ζητούμενο 2 – Κρυπτογραφημένο Chat μέσω TCP / IP sockets

Η ουσιαστική αλλαγή που πραγματοποιείται στο Ζητούμενο 1 είναι πως πλέον δεν αποστέλλεται κατευθείαν το κείμενο που θέλει κάποιος να στείλει στο άλλο άκρο της συνομιλίας, αλλά η κρυπτογραφημένη μορφή αυτού. Αυτό επιτυγχάνεται μέσω της συσκευής cryptodev. Το κλειδί της κρυπτογράφησης κατά σύμβαση είναι hard coded και στα δύο άκρα της συνομιλίας.

3.1. Πηγαίος κώδικας (source code)

3.1.1. crypto-server.c

```
int main(void)
{
    int sockfd, newsockfd;

    char recv_buf[BUFSIZE], send_buf[BUFSIZE];
    char addrstr[INET_ADDRSTRLEN];

    ssize_t n;
    socklen_t len;
    struct sockaddr_in server_addr, client_addr;

    fd_set master;

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

    /* crypto declarations*/
    int i;
    int crypto_fd;
    struct session_op sess;
    struct crypt_op cryp;
    struct {
        unsigned char   in[DATA_SIZE],
                encrypted[DATA_SIZE],
                decrypted[DATA_SIZE];
    } data;
    unsigned char *iv = "1234567890qwerty";
    unsigned char *key = "1234567890qwerty";

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

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

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

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

    /* Listen for incoming connections */
    if (listen(sockfd, 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 ((newsockfd = accept(sockfd, (struct sockaddr *)&client_addr, &len))
< 0) {
            perror("accept");
            exit(1);
        }
        if (!inet_ntop(AF_INET, &client_addr.sin_addr, addrstr,
sizeof(addrstr))) {
            perror("could not format IP address");
            exit(1);
        }
        fprintf(stderr, "Incoming connection from %s:%d\n", addrstr,
ntohs(client_addr.sin_port));

        printf("\nNew session started");

        /* getting file descriptor for cryptographic device*/
        crypto_fd = open("/dev/crypto", O_RDWR);
        if (crypto_fd < 0) {
            perror("open(/dev/crypto)");
            return 1;
        }
        else {
            printf("...and no one can hear you!\n\n");
        }


        /* We break out of the loop when the remote peer goes away */
        for (;;) {
            FD_ZERO(&master);
            FD_SET(newsockfd, &master);
            FD_SET(0, &master);
            fflush(stdout);
            int readsockets = select(newsockfd + 1, &master, NULL, NULL, NULL);

            if (readsockets < 0) {
```

```
        perror("select");
        exit(1);
    }
    else {
        /* client input */
        if (FD_ISSET(newsockfd, &master)){

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

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

            /* read incoming data*/
            n = read(newsockfd, recv_buf, sizeof(recv_buf));
            if (n <= 0) {
                if (n < 0)
                    perror("read from remote peer failed");
                else
                    fprintf(stderr, "\nPeer went away...\n");
                break;
            }
            fprintf(stdout, "\nRemote: ");
            fflush(stdout);

            /*Decrypt received to data.decrypted*/
            cryp.ses = sess.ses;
            cryp.len = sizeof(recv_buf);
            cryp.src = recv_buf;
            cryp.dst = data.decrypted;
            cryp.iv = iv;
            cryp.op = COP_DECRYPT;

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

            /* print decrypted data */
            for (i = 0; i < n; i++) {
                if (data.decrypted[i] == '\n')
                    break;
                else
                    printf("%c", data.decrypted[i]);
            }
            printf("\n");
        }
        else{
            /* server output */

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

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

            /* read from stdin*/
            n = read(0, send_buf, sizeof(send_buf));
            if (n <= 0) {
                if (n < 0)
```

```
                        perror("read from localy  failed");
                    else
                        fprintf(stderr, "I went away\n");
                    break;
                }

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

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

                if (insist_write(newsockfd, data.encrypted,
sizeof(data.encrypted)) != sizeof(data.encrypted)) {
                    perror("write to remote failed");
                    break;

                }
                fflush(stdout);
            }
        }
        /* Finish crypto session */
        if (ioctl(crypto_fd, CIOCFSESSION, &sess.ses)) {
            perror("ioctl(CIOCFSESSION)");
            return 1;
        }
    }
    if (close(newsockfd) < 0) {
        perror("close");
    }

    if (close(crypto_fd) < 0) {
        perror("close(crypto_fd)");
        return 1;
    }
}

    return 1;
}
```

3.1.2. crypto-client.c

```
int main(int argc, char *argv[])
{
    int sd, port;
    ssize_t n;
    char send_buf[BUFSIZE], recv_buf[BUFSIZE];
    char *hostname;
    struct hostent *hp;
    struct sockaddr_in sa;

    fd_set master;

    /* crypto declarations*/
    int i;
    int crypto_fd;
    struct session_op sess;
    struct crypt_op cryp;
    struct {
        unsigned char   in[DATA_SIZE],
```

```
            encrypted[DATA_SIZE],
            decrypted[DATA_SIZE];
} data;
unsigned char *iv = "1234567890qwerty";
unsigned char *key = "1234567890qwerty";

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

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. New session started");

/* getting file descriptor for cryptographic device*/
crypto_fd = open("/dev/crypto", O_RDWR);
if (crypto_fd < 0) {
    perror("open(/dev/crypto)");
    return 1;
}
else {
    printf("...and no one can hear you!\n\n");
}

/* Read answer and write it to standard output */
for (;;) {

    FD_ZERO(&master);
    FD_SET(sd, &master);
    FD_SET(0, &master);
    fflush(stdout);
    int readsockets = select(sd + 1, &master, NULL, NULL, NULL);

    if (readsockets < 0) {
        perror("select");
        exit(EXIT_FAILURE);
    }
    else{
        /* server output */
        if (FD_ISSET(sd, &master)){
```

```
/*Get crypto session for AES128*/
sess.cipher = CRYPTO_AES_CBC;
sess.keylen = KEY_SIZE;
sess.key = key;

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

/* read incoming data*/
n = read(sd, recv_buf, sizeof(recv_buf));
if (n <= 0) {
    if (n < 0)
        perror("read from remote peer failed");
    else
        fprintf(stderr, "\nPeer went away...\n");
        break;
    }
    fprintf(stdout, "\nRemote: ");
    fflush(stdout);

    /*Decrypt received to data.decrypted*/
    cryp.ses = sess.ses;
    cryp.len = sizeof(recv_buf);
    cryp.src = recv_buf;
    cryp.dst = data.decrypted;
    cryp.iv = iv;
    cryp.op = COP_DECRYPT;

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

    /* print decrypted data */
    for (i = 0; i < n; i++) {
        if (data.decrypted[i] == '\n')
            break;
        else
            printf("%c", data.decrypted[i]);
    }
    printf("\n");
}
else{
    /* client input */


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

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

    /* read from stdin*/
    n = read(0, send_buf, sizeof(send_buf));
    if (n <= 0) {
        if (n < 0)
            perror("read from localy  failed");
        else
            fprintf(stderr, "I went away\n");
        break;
    }
```

```
                    /*Encrypt writen data to data.encrypted*/
                    cryp.ses = sess.ses;
                    cryp.len = sizeof(send_buf);
                    cryp.src = send_buf;
                    cryp.dst = data.encrypted;
                    cryp.iv = iv;
                    cryp.op = COP_ENCRYPT;

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

                    if (insist_write(sd, data.encrypted,
sizeof(data.encrypted)) != sizeof(data.encrypted)) {
                        perror("write to client failed");
                        break;

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

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

    if (close(crypto_fd) < 0) {
        perror("close(crypto_fd)");
        return 1;
    }

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

3.2. Περιγραφή υλοποίησης των επιμέρους τμημάτων του παραπάνω κώδικα

3.2.1. Προσθήκες στις συναρτήσεις main() του client και του server - Αρχικοποίηση cryptodev

```
    int crypto_fd;
    struct session_op sess;
    struct crypt_op cryp;
    struct {
        unsigned char   in[DATA_SIZE],
                encrypted[DATA_SIZE],
                decrypted[DATA_SIZE];
    } data;
    unsigned char *iv = "1234567890qwerty";
    unsigned char *key = "1234567890qwerty";

    /* getting file descriptor for cryptographic device*/
    crypto_fd = open("/dev/crypto", O_RDWR);
    if (crypto_fd < 0) {
        perror("open(/dev/crypto)");
        return 1;
```

```
}
else {
    printf("...and no one can hear you!\n\n");
}
```

Στη `main()` του server και του client αρχικοποιούμε τα session για κρυπτογράφηση μέσω cryptodev, ένα struct data με πίνακες χαρακτήρων όπου θα κρατάμε το αρχικό, το κρυπτογραφημένο και το αποκρυπτογραφημένο μήνυμα και τέλος ανοίγουμε τη συσκευή με την κλήση της `open("/dev/crypto", O_RDWR)` η οποία επιστρέφει έναν file descriptor που δείχνει σε αυτή.

3.2.2. Εκκίνηση Session (CIOCGSESSION)

```
struct session_op sess;
memset(&sess, 0, sizeof(sess));
/*Get crypto session for AES128*/
sess.cipher = CRYPTO_AES_CBC;
sess.keylen = KEY_SIZE;
sess.key = key;
if (ioctl(crypto_fd, CIOCGSESSION, &sess)) {
  perror("ioctl(CIOCGSESSION)");
  return 1;
}
```

Με την κλήση της `ioctl(crypto_fd, CIOCGSESSION, &sess)` ξεκινάμε μια cryptodev session, την οποία θα χρησιμοποιήσουμε σε όλες τις λειτουργίες κρυπτογράφησης και αποκρυπτογράφησης. Ρυθμίζουμε αυτή τη session ώστε να πραγματοποιηθεί με αλγόριθμο AES128.

3.2.3. Κρυπτογράφηση (CIOCCRYPT, cryp->op = COP_ENCRYPT)

```
/*Encrypt writen data to data.encrypted*/
cryp.ses = sess.ses;
cryp.len = sizeof(send_buf);
cryp.src = send_buf;
cryp.dst = data.encrypted;
cryp.iv = iv;
cryp.op = COP_ENCRYPT;

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

Η κλήση της `ioctl(crypto_fd, CIOCCRYPT, &cryp)` πραγματοποιεί κρυπτογράφηση του μηνύματος που γράψαμε στο stdin και που αντιγράψαμε στο πεδίο `cryp.src` ώστε να την εναποθέτει στο πεδίο `cryp.dst`. Στη συνέχεια αποστέλλεται όπως πριν στον απομακρυσμένο χρήστη.

3.2.4. Αποκρυπτογράφηση (CIOCCRYPT, cryp->op = COP_DENCRYPT)

```
/*Decrypt received to data.decrypted*/
cryp.ses = sess.ses;
cryp.len = sizeof(recv_buf);
cryp.src = recv_buf;
cryp.dst = data.decrypted;
cryp.iv = iv;
cryp.op = COP_DECRYPT;

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

Η ίδια εντολή με διαφορετικό operation αυτή τη φορά (`cryp.op =`

COP_DECRYPT) πραγματοποιεί αποκρυπτογράφηση. Ομοίως και εδώ αντιγράφουμε το μήνυμα που λαμβάνουμε από τον απομακρυσμένο χρήστη στο στο πεδίο cryp.src ώστε να την καταλήξει η αποκρυπτογράφηση αυτού στο πεδίο cryp.dst. Στη συνέχεια εκτυπώνεται στο stdout.

3.2.5. Τερματισμός Session (CIOCFSESSION)

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

Με την κλήση της ioctl(crypto_fd, CIOCFSESSION, &sess.ses) κλείνουμε την session.

4. Ζητούμενο 3 – Κρυπτογραφική συσκευή VirtIO για QEMU-KVM

Όπως είπαμε και στην εισαγωγή θέλουμε να κρυπτογραφήσουμε τα μηνύματα μας, τα οποία θα στέλνονται αυτή τη φορά από εφαρμογή που θα τρέχει στο userspace ενός VM. Αυτή η διαδικασία θα ήταν αργή αν αυτό. συνέβαινε σε περιβάλλον πλήρης εικονικοποίησης (full virtualization). Για αυτό το λόγο υλοποιούμε έναν paravirtualized οδηγό συσκευής χαρακτήρων virtio-crypto οποίος χωρίζεται σε δύο μέρη, το front end που εισάγεται στον πυρήνα του guest και το backend που εισάγεται στον κώδικα του QEMU (δηλαδή του διαχειριστή -hypervisor- του VM). Αυτή η συσκευή θα αναλάβει την επικοινωνία με την πραγματική συσκευή κρυπτογράφησης cryptodev που χρησιμοποιήσαμε στο Ζητούμενο 2, η οποία και εισάγεται ως module στον kernel του host. Παρακάτω εξετάζουμε τον κώδικα του front και του back end χωριστά.

4.1. Πηγαίος κώδικας (source code)

4.1.1. Front-end

```
/************************************
 * 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;

    //Extra declarations
    unsigned int num_out = 0;
    unsigned int num_in = 0;
    struct scatterlist syscall_type_sg , host_fd_sg, *sgs[2];

    debug("Entering open");

    /* Allocate all data that will be sent to the host. */
    syscall_type = kmalloc(sizeof(*syscall_type), GFP_KERNEL);
    *syscall_type = VIRTIO_CRYPTO_SYSCALL_OPEN;
```

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

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

    /* Wait for the host to process our data. */
    if (down_interruptible(&crdev->lock)) {
        return -ERESTARTSYS;
    }
    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);

    up(&crdev->lock);

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

fail:
    debug("Leaving open");
    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;
    int *host_fd;
    //unsigned int syscall_type = VIRTIO_CRYPTO_SYSCALL_CLOSE;

    //Extra declarations
```

```
    struct scatterlist syscall_type_sg, host_fd_sg, *sgs[2];
    unsigned int len;
    unsigned int num_out = 0;
    unsigned int num_in = 0;
    long *host_return_val;

    debug("Entering release");

    /* Allocate all data that will be sent to the host. */
    syscall_type = kmalloc(sizeof(*syscall_type), GFP_KERNEL);
    *syscall_type = VIRTIO_CRYPTO_SYSCALL_CLOSE;
    host_fd = kmalloc(sizeof(*host_fd), GFP_KERNEL);
    *host_fd = crof->host_fd;

    /* Send data to the host. */
    //syscall_type
    sg_init_one(&syscall_type_sg, syscall_type, sizeof(*syscall_type));
    sgs[num_out++] = &syscall_type_sg;

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

    /* Wait for the host to process our data. */
    if (down_interruptible(&crdev->lock)) {
        return -ERESTARTSYS;
    }
    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);

    up(&crdev->lock);

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

static long crypto_chrdev_ioctl(struct file *filp, unsigned int cmd,
                                unsigned long arg)
{
    long ret = 0;
    int err;
    struct crypto_open_file *crof = filp->private_data;
    struct crypto_device *crdev = crof->crdev;
    struct virtqueue *vq = crdev->vq;

    //Extra declarations
    struct scatterlist syscall_type_sg, host_fd_sg, cmd_sg, session_sg,
crypto_sg,/*output_msg_sg, input_msg_sg,*/ key_msg_sg, ret_sg, dest_msg_sg,
src_msg_sg, iv_msg_sg, *sgs[9];
    unsigned int num_out, num_in, len;
#define MSG_LEN 100
    unsigned char *output_msg, *input_msg, *key, *src, *dst, *iv;
    unsigned int *syscall_type;
    unsigned int *ioctl_cmd;
    int *host_fd;
    long *host_return_val;
    uint32_t *ses;
    struct session_op seop, *seop_pointer;
    struct crypt_op crop, *crop_pointer;

    debug("Entering ioctl frontend");

    /* Allocate all data that will be sent to the host. */
    output_msg = kmalloc(MSG_LEN, GFP_KERNEL);
```

```
    input_msg = kmalloc(MSG_LEN, GFP_KERNEL);
    syscall_type = kmalloc(sizeof(*syscall_type), GFP_KERNEL);
    *syscall_type = VIRTIO_CRYPTO_SYSCALL_IOCTL;
    host_fd = kmalloc(sizeof(*host_fd), GFP_KERNEL);
    *host_fd = crof->host_fd;
    printk("crof->host_fd = %d \n", *host_fd);
    ioctl_cmd = kmalloc(sizeof(*ioctl_cmd), GFP_KERNEL);
    *ioctl_cmd = cmd;
    host_return_val = kmalloc(sizeof(*host_return_val), GFP_KERNEL);
    *host_return_val = -1;
    ses = kmalloc(sizeof(*ses), GFP_KERNEL);
    *ses = 0;

    num_out = 0;
    num_in = 0;
    key = NULL;
    src = NULL;
    dst = NULL;
    iv = NULL;

    /* These are common to all ioctl commands. */
    //syscall_type  | num_out++ = 1
    sg_init_one(&syscall_type_sg, syscall_type, sizeof(*syscall_type));
    sgs[num_out++] = &syscall_type_sg;
    //host_fd  | num_out++ = 2
    sg_init_one(&host_fd_sg, host_fd, sizeof(*host_fd));
    sgs[num_out++] = &host_fd_sg;
    debug("After sg_initS and before switch command execution");
    printk("cmd = %d \n", cmd);

    /* Add all the cmd specific sg lists. */
    switch (cmd) {
    case CIOCGSESSION:
        debug("Entering CIOCGSESSION frontend");
        memcpy(output_msg, "Hello HOST from ioctl CIOCGSESSION.", 36);
        input_msg[0] = '\0';

        //cmd ={CIOCGSESSION, CIOCFSESSION, CIOCCRYPT}  | num_out++ = 3
        sg_init_one(&cmd_sg, ioctl_cmd, sizeof(*ioctl_cmd));
        sgs[num_out++] = &cmd_sg;

        //get session struct from arg from userspace
        if (unlikely(copy_from_user(&seop, (struct session_op*)arg,
sizeof(struct session_op))))
            return -EFAULT;

        //key
        key = kmalloc(seop.keylen, GFP_KERNEL);
        if (unlikely(copy_from_user(key, seop.key, seop.keylen)))
            return -EFAULT;

        //seop->key should be sent to backend  | num_out++ = 4
        sg_init_one(&key_msg_sg, key, seop.keylen);
        sgs[num_out++] = &key_msg_sg;

        seop_pointer = &seop;
        //session struct seop should be filled from backend  | num_out +
num_in++ = 5
        sg_init_one(&session_sg, seop_pointer, sizeof(struct session_op *));
        sgs[num_out + num_in++] = &session_sg;

        //host_return_val  | num_out + num_in++ = 6
        sg_init_one(&ret_sg, host_return_val, sizeof(*host_return_val));
        sgs[num_out + num_in++] = &ret_sg;

        debug("Leaving CIOCGSESSION frontend");
        break;
```

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

        //cmd ={CIOCGSESSION, CIOCFSESSION, CIOCCRYPT}  | num_out++ = 3
        sg_init_one(&cmd_sg, ioctl_cmd, sizeof(*ioctl_cmd));
        sgs[num_out++] = &cmd_sg;

        if (unlikely(copy_from_user(ses, (uint32_t*)arg, sizeof(*ses))))
            return -EFAULT;

        // session->ses should be sent to backend  | num_out++ = 4
        sg_init_one(&session_sg, ses, sizeof(*ses));
        sgs[num_out++] = &session_sg;

        //host_return_val  | num_out + num_in++ = 5
        sg_init_one(&ret_sg, host_return_val, sizeof(*host_return_val));
        sgs[num_out + num_in++] = &ret_sg;

        debug("Leaving CIOCFSESSION frontend");
        break;

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

        //cmd ={CIOCGSESSION, CIOCFSESSION, CIOCCRYPT}  | num_out++ = 3
        sg_init_one(&cmd_sg, ioctl_cmd, sizeof(*ioctl_cmd));
        sgs[num_out++] = &cmd_sg;

        // crypto struct crop should be sent to backend
        if (unlikely(copy_from_user(&crop, (struct crypt_op*)arg, sizeof(struct
crypt_op))))
            return -EFAULT;
        crop_pointer = &crop;
        sg_init_one(&crypto_sg, crop_pointer, sizeof(struct crypt_op*));
        sgs[num_out++] = &crypto_sg;

        // crop->src should be sent to backend
        src = kmalloc(crop.len, GFP_KERNEL);
        if (unlikely(copy_from_user(src, crop.src, crop.len)))
            return -EFAULT;
        sg_init_one(&src_msg_sg, src, crop.len);
        sgs[num_out++] = &src_msg_sg;

        iv = kmalloc(16, GFP_KERNEL);
        // crop->iv should be sent to backend
        if (unlikely(copy_from_user(iv, crop.iv, 16)))
            return -EFAULT;
        sg_init_one(&iv_msg_sg, iv, 16);
        sgs[num_out++] = &iv_msg_sg;

        // crop->dst should filled from backend
        dst = kmalloc(crop.len, GFP_KERNEL);
        sg_init_one(&dest_msg_sg, dst, crop.len);
        sgs[num_out + num_in++] = &dest_msg_sg;

        //host_return_val  | num_out + num_in++ = ?
        sg_init_one(&ret_sg, host_return_val, sizeof(*host_return_val));
        sgs[num_out + num_in++] = &ret_sg;

        debug("Leaving CIOCCRYPT frontend");
        break;

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

```
        break;
    }

    /* Wait for the host to process our data. */
    if (down_interruptible(&crdev->lock)) {
        return -ERESTARTSYS;
    }
    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 */;
    up(&crdev->lock);

    /* AFTER SGS ARE READY FROM BACKEND */
    switch(cmd) {
    case (CIOCGSESSION):
        debug("CIOCGSESSION RET");
        if (unlikely(copy_to_user((struct session_op*)arg, &seop, sizeof(struct
session_op))))
            return -EFAULT;
        kfree(key);

        break;

    case (CIOCFSESSION):
        debug("CIOCFSESSION RET");

        break;

    case (CIOCCRYPT):
        debug("CIOCCRYPT RET");
        if (unlikely(copy_to_user((struct crypt_op*)arg, &crop, sizeof(struct
crypt_op))))
            return -EFAULT;
        if (unlikely(copy_to_user(crop.dst, dst, crop.len*sizeof(char))))
            return -EFAULT;
        kfree(src);
        kfree(dst);
        kfree(iv);

        break;
    }

    kfree(output_msg);
    kfree(input_msg);
    kfree(syscall_type);

    debug("Leaving ioctl front end");

    return ret;
}
```

4.1.2. Back-end

```
static void vq_handle_output(VirtIODevice *vdev, VirtQueue *vq)
{
    VirtQueueElement elem;
    unsigned int *syscall_type;
    int *host_fd;

    DEBUG_IN();

    if (!virtqueue_pop(vq, &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_CRYPTO_SYSCALL_TYPE_OPEN:
        DEBUG("VIRTIO_CRYPTO_SYSCALL_TYPE_OPEN");
        /* We get the file descriptor the frondend sends and
        we return there the result of open syscall */

        host_fd = elem.in_sg[0].iov_base;
        *host_fd = open(CRYPTODEV_FILENAME, O_RDWR);
        printf("*host_fd after open= %d\n", *host_fd);

        break;

    case VIRTIO_CRYPTO_SYSCALL_TYPE_CLOSE:
        DEBUG("VIRTIO_CRYPTO_SYSCALL_TYPE_CLOSE");
        /* We get the file descriptor the frondend sends but
        there is no need to send anything back. We simply
        close the file  */

        host_fd = elem.out_sg[1].iov_base;
        printf("*host_fd before close= %d\n", *host_fd);
        close(*host_fd);

        break;

    case VIRTIO_CRYPTO_SYSCALL_TYPE_IOCTL:
        DEBUG("VIRTIO_CRYPTO_SYSCALL_TYPE_IOCTL");
        /* First we obtain the adress of the message we need to
        encrypt and the one we need to write to. */
        unsigned char *output_msg ;
        unsigned char *input_msg ;
        unsigned int *ioctl_cmd;
        long * host_return_val;

        host_fd = elem.out_sg[1].iov_base;
        ioctl_cmd = elem.out_sg[2].iov_base;
        printf("Host_fd is: %d\n", *host_fd);
        switch (*ioctl_cmd) {
        case(CIOCGSESSION):
            DEBUG("Entering CIOCGSESSION backend");

            unsigned char *session_key;
            unsigned char *temp1;
            struct session_op *session_op;

            //read from here
            session_key = elem.out_sg[3].iov_base;
            session_op = elem.in_sg[0].iov_base;
            //write here
            host_return_val = elem.in_sg[1].iov_base;
            //we save the key we recieved so that the frondend gets the right
address
            temp1 = session_op->key;
            session_op->key = session_key;
            //syscall
            *host_return_val = ioctl(*host_fd, CIOCGSESSION, session_op);
            session_op->key = temp1;

            DEBUG("Leaving CIOCGSESSION backend");
            break;

        case(CIOCFSESSION):
            DEBUG("Entering CIOCFSESSION backend");
```

```
            uint32_t *ses_id;
            //read from here
            ses_id = elem.out_sg[3].iov_base;
            //write here
            host_return_val = elem.in_sg[0].iov_base;
            // syscall
            *host_return_val = ioctl(*host_fd, CIOCFSESSION, ses_id);

            DEBUG("Leaving CIOCFSESSION backend");
            break;

        case(CIOCCRYPT):
            DEBUG("Entering CIOCCRYPT backend");

            struct crypt_op crypt_op;

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

            host_return_val = elem.in_sg[1].iov_base;
            *host_return_val = ioctl(*host_fd, CIOCCRYPT, &crypt_op);

            DEBUG("Leaving CIOCCRYPT backend");
            break;

        }
        break;

    default:
        DEBUG("Unknown syscall_type");
    }
    virtqueue_push(vq, &elem, 0);
    virtio_notify(vdev, vq);
}
```

4.2. Περιγραφή υλοποίησης των επιμέρους τμημάτων του παραπάνω κώδικα του front end (guest)

Αυτό το κομμάτι της υλοποίησης αποτελεί τον character device driver του virtio-crypto device στην οποία θα απευθύνεται η εφαρμογή που θα τρέχει στο userspace του VM, για κρυπτογράφηση. Θα υλοποιεί τις λειτουργίες `open`, `release` και `ioctl`. Πιο συγκεκριμένα θα τοποθετεί δείκτες σε buffers μέσα σε scatterlists και θα φροντίζει ώστε να ενημερώνουν το back end κομμάτι της εφαρμογής για αυτούς.

Σημείωση: Φροντίσουμε, καθ' υπόδειξη των βοηθών, οι μεταβλητές που μεταφέρονται να είναι heap allocated. Για αυτό και οι δηλώσεις των μεταβλητών είναι της μορφής:

```
unsigned int *syscall_type;
syscall_type = kmalloc(...);
sg_init_one(&syscall_type_sg, syscall_type,
                    sizeof(*syscall_type));
```

4.2.1. Lock στο crypto.h και init στο crypto-module.c

```
/* Device info.*/
struct crypto_device {
    /* ... */
    /* ?? Lock ?? */
    struct semaphore lock;
    /* ... */
};
```

Στο παραπάνω struct προσθέσαμε έναν struct semaphore lock τον οποίο αρχικοποιούμε μέσα στη συνάρτηση `static int virtcons_probe()` του crypto-module.c, η οποία και καλείται μόλις ο πυρήνας ανιχνεύει μια νέα συσκευή virtio. Τη χρησιμότητα αυτού του lock θα την εξηγήσουμε παρακάτω.

4.2.2. System Call Open

```
unsigned int *syscall_type;
int *host_fd;

//Extra declarations
unsigned int num_out = 0;
unsigned int num_in = 0;
struct scatterlist syscall_type_sg , host_fd_sg, *sgs[2];

/* Allocate all data that will be sent to the host. */
syscall_type = kmalloc(sizeof(*syscall_type), GFP_KERNEL);
*syscall_type = VIRTIO_CRYPTO_SYSCALL_OPEN;
host_fd = kmalloc(sizeof(*host_fd), GFP_KERNEL);
*host_fd = -1;

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

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

/* Wait for the host to process our data. */
if (down_interruptible(&crdev->lock)) {
    return -ERESTARTSYS;
}
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);

up(&crdev->lock);

/* If host failed to open() return -ENODEV. */
if ((crof->host_fd = *host_fd) <= 0) {
    ret= -ENODEV;
    goto fail;
}
```

Εκχωρούμε στο `syscalltype VIRTIO_CRYPTO_SYSCALL_OPEN` και βάζουμε τη διεύθυνσή του δείκτη σε ένα scatterlist. Επιπλέον πρέπει να αρχικοποιηθεί μία scatterlist, στην οποία το back-end θα γράψει το αποτέλεσμα της open που θα γίνει στο `/dev/crypto` του host. Βάζουμε τις δυο αυτές scatterlists σε ένα array (sgs) το οποίο προσθέτουμε στο `virtqueue (virtqueue_add_sgs)` και μέσω της κλήσης `virtqueue_kick(crdev->vq)` θα ειδοποιηθεί το back end για να κάνει τη διαχείρισή τους. Μέχρι να ειδοποιήσει το front end (notify) από το back-end πως το τελευταίο ολοκλήρωσε την επεξεργασία τους, η εκτέλεση μπλοκάρει στο while loop.

Σημείωση: Όλες οι virtqueue operations είναι απαραίτητο να γίνονται μέσα σε κρίσιμο κομμάτι κώδικα, καθώς αν δυο διεργασίες πραγματοποιήσουν ταυτόχρονα operations τέτοιου είδους, το αποτέλεσμα μπορεί να είναι απροσδιόριστο και διαφορετικό από το αναμενόμενο. Ενδέχεται δηλαδή, κάποια διεργασία να παραλάβει τα scatterlists άλλης και να υπάρξει data corruption, ή η άλλη διεργασία να μην παραλάβει τα δεδομένα της και έτσι παραμείνει για πάντα στο while loop της `virtqueue_get_buf`. Εκεί έγκειται

και η χρησιμότητα του lock που προσθέσαμε και αναφέραμε παραπάνω.

4.2.3. System Call Release

```
unsigned int *syscall_type;
int *host_fd;
//Extra declarations
struct scatterlist syscall_type_sg, host_fd_sg, *sgs[2];
unsigned int len;
unsigned int num_out = 0;
unsigned int num_in = 0;
//Allocate all data that will be sent to the host.
syscall_type = kmalloc(sizeof(*syscall_type), GFP_KERNEL);
*syscall_type = VIRTIO_CRYPTO_SYSCALL_CLOSE;
host_fd = kmalloc(sizeof(*host_fd), GFP_KERNEL);
*host_fd = crof->host_fd;

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

/* Wait for the host to process our data. */
if (down_interruptible(&crdev->lock)) {
    return -ERESTARTSYS;
}
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);

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

Ομοίως και για την κλήση close πρέπει να αποσταλεί ο τύπος της λειτουργίας (VIRTIO_CRYPTO_SYSCALL_CLOSE), καθώς και ο file descriptor στον οποίο θα πραγματοποιηθεί η close. Ο file descriptor αυτός (του host) είναι αποθηκευμένος σε δομή στο private data πεδίο του file structure στο οποίο καλείται η close.

4.2.4. System Call Ioctl

Κοινός κώδικας και των τριών ioctl υποπεριπτώσεων.

```
    struct scatterlist syscall_type_sg, host_fd_sg, cmd_sg, session_sg,
crypto_sg, key_msg_sg, ret_sg, dest_msg_sg, src_msg_sg, iv_msg_sg,
*sgs[9];
    unsigned int num_out, num_in, len;
    unsigned int *syscall_type;
    unsigned int *ioctl_cmd;
    int *host_fd;
    long *host_return_val;
    uint32_t *ses;
    struct session_op seop, *seop_pointer;
    struct crypt_op crop, *crop_pointer;

    //Allocate all data that will be sent to the host.
    syscall_type = kmalloc(sizeof(*syscall_type), GFP_KERNEL);
    *syscall_type = VIRTIO_CRYPTO_SYSCALL_IOCTL;
    host_fd = kmalloc(sizeof(*host_fd), GFP_KERNEL);
    *host_fd = crof->host_fd;
    printk("crof->host_fd = %d \n", *host_fd);
```

```
ioctl_cmd = kmalloc(sizeof(*ioctl_cmd), GFP_KERNEL);
*ioctl_cmd = cmd;
host_return_val = kmalloc(sizeof(*host_return_val), GFP_KERNEL);
*host_return_val = -1;
ses = kmalloc(sizeof(*ses), GFP_KERNEL);
*ses = 0;

num_out = 0;
num_in = 0;
key = NULL;
src = NULL;
dst = NULL;
iv = NULL;

// These are common to all ioctl commands
//syscall_type  | num_out++ = 1
sg_init_one(&syscall_type_sg, syscall_type, sizeof(*syscall_type));
sgs[num_out++] = &syscall_type_sg;
//host_fd  | num_out++ = 2
sg_init_one(&host_fd_sg, host_fd, sizeof(*host_fd));
sgs[num_out++] = &host_fd_sg;
//cmd ={CIOCGSESSION, CIOCFSESSION, CIOCCRYPT}  | num_out++ = 3
sg_init_one(&cmd_sg, ioctl_cmd, sizeof(*ioctl_cmd));
sgs[num_out++] = &cmd_sg;
```

Για την κλήση της ioctl στέλνουμε στο back end την τιμή `VIRTIO_CRYPTO_SYSCALL_IOCTL`, καθώς και το command του ioctl που θέλουμε να πραγματοποιηθεί. Ακολούθως, ανάλογα με το command, αποστέλλονται και διαφορετικές scatterlists που περιγράφονται παρακάτω. Το πλήθος και η σημασία των scatterlist που αποστέλλονται μεταβάλλονται ανάλογα με την περίπτωση, όμως η διαδικασία προσθήκης στη virtqueue, αποστολής και λήψης απάντησης (`virtqueue_add_sgs, virtqueue_kick, virtqueue_get_buf`) παραμένει ίδια.

4.2.4.1. CIOCGSESSION

```
//get session struct from arg from userspace
if (unlikely(copy_from_user(&seop, (struct session_op*)arg,
                    sizeof(struct session_op)))) {
    return -EFAULT;
}
//key
key = kmalloc(seop.keylen, GFP_KERNEL);
if (unlikely(copy_from_user(key, seop.key, seop.keylen)))
    return -EFAULT;
//seop->key should be sent to backend  | num_out++ = 4
sg_init_one(&key_msg_sg, key, seop.keylen);
sgs[num_out++] = &key_msg_sg;

seop_pointer = &seop;
//session struct seop should be filled
//from backend  | num_out + num_in++ = 5
sg_init_one(&session_sg, seop_pointer, sizeof(struct session_op *));
sgs[num_out + num_in++] = &session_sg;

//host_return_val  | num_out + num_in++ = 6
sg_init_one(&ret_sg, host_return_val, sizeof(*host_return_val));
sgs[num_out + num_in++] = &ret_sg;
```

Για το `ioctl` command `CIOCGSESSION`, χρειάζονται επιπλέον δύο scatterlists. Η πρώτη, θα είναι για ανάγνωση από το front end και θα περιέχει το κλειδί της κρυπτογράφησης. Η δεύτερη θα είναι για εγγραφή από το back end και θα είναι το `struct session_op` της κρυπτογράφησης. Όσον αφορά τη δεύτερη, τα περισσότερα

πεδία του struct έχουν ήδη γραφτεί από τον χρήστη του VM και παρότι το struct είναι προς εγγραφή πρέπει να σταλούν στο back end ως έχουν, αφού είναι πεδία απαραίτητα για την κρυπτογράφηση.

Πριν αρχικοποιηθούν οι scatterlists θα πρέπει να πάρουμε από τον χρήστη του VM τα πεδία των struct που χρειάζονται μέσω της παραμέτρου `arg`. Αρχικά θα πρέπει να διαβάσουμε από το userspace το struct session_op. Στη συνέχεια στο πεδίο του `session_op, key`, βρίσκεται η διεύθυνση στο user space που είναι αποθηκευμένο το κλειδί. Στο πεδίο `keylen` βρίσκεται το μήκος του κλειδιού. Γνωρίζοντας αυτές τις δυο πληροφορίες αντιγράφουμε και το κλειδί της κρυπτογράφησης από το user space του VM. Η αντιγραφή γίνεται μέσω του `copy_from_user`, το οποίο, με ασφαλή τρόπο, αντιγράφει δεδομένα από διευθύνσεις user space στο kernel space.

4.2.4.2. CIOCFSESSION

```
if (unlikely(copy_from_user(ses, (uint32_t*)arg, sizeof(*ses))))
    return -EFAULT;

// session->ses should be sent to backend  | num_out++ = 4
sg_init_one(&session_sg, ses, sizeof(*ses));
sgs[num_out++] = &session_sg;

//host_return_val  | num_out + num_in++ = 5
sg_init_one(&ret_sg, host_return_val, sizeof(*host_return_val));
sgs[num_out + num_in++] = &ret_sg;
```

Για το `ioctl` command `CIOCFSESSION`, χρειάζεται μία scatterlist, η οποία θα είναι για ανάγνωση από το back end. Αυτή περιέχει το πεδίο `session_op.ses`, το οποίο θα πρέπει να αντιγραφεί από το user space του VM μέσω της παραμέτρου arg. Για την επιτυχή αντιγραφή χρησιμοποιούμε το `copy_from_user`.

4.2.4.3. CIOCCRYPT

```
// crypto struct crop should be sent to backend
if (unlikely(copy_from_user(&crop, (struct crypt_op*)arg, sizeof(struct
crypt_op))))
    return -EFAULT;
crop_pointer = &crop;
sg_init_one(&crypto_sg, crop_pointer, sizeof(struct crypt_op*));
sgs[num_out++] = &crypto_sg;

// crop->src should be sent to backend
src = kmalloc(crop.len, GFP_KERNEL);
if (unlikely(copy_from_user(src, crop.src, crop.len)))
    return -EFAULT;
sg_init_one(&src_msg_sg, src, crop.len);
sgs[num_out++] = &src_msg_sg;

iv = kmalloc(16, GFP_KERNEL);
// crop->iv should be sent to backend
if (unlikely(copy_from_user(iv, crop.iv, 16)))
    return -EFAULT;
sg_init_one(&iv_msg_sg, iv, 16);
sgs[num_out++] = &iv_msg_sg;

// crop->dst should filled from backend
dst = kmalloc(crop.len, GFP_KERNEL);
sg_init_one(&dest_msg_sg, dst, crop.len);
sgs[num_out + num_in++] = &dest_msg_sg;

//host_return_val  | num_out + num_in++ = ?
sg_init_one(&ret_sg, host_return_val, sizeof(*host_return_val));
sgs[num_out + num_in++] = &ret_sg;
```

Για το `ioctl` command `CIOCCRYPT`, χρειάζονται επιπλέον τέσσερα scatterlists. Η πρώτη, θα είναι για ανάγνωση από το back-end και θα περιέχει το struct crypto_op της κρυπτογράφησης. Η δεύτερη θα είναι επίσης για ανάγνωση και θα περιέχει το πεδίο `crypto_op.src` της κρυπτογράφησης. Η τρίτη θα περιέχει το iv pointer του crypto_op και θα είναι επίσης για ανάγνωση από το back end. Η τέταρτη και τελευταία θα είναι για εγγραφή από το back end και θα είναι το πεδίο `crypto_op.dst` της κρυπτογράφησης. Πριν αρχικοποιηθούν οι scatterlists θα πρέπει να πάρουμε από τον χρήστη του VM τα πεδία που χρειάζονται μέσω της παραμέτρου arg του ioctl. Αρχικά θα πρέπει να διαβάσουμε από το userspace το struct `crypto_op`, από τη διεύθυνση `arg`. Στη συνέχεια στο πεδίο του `struct_op.src`, βρίσκεται η διεύθυνση στο user space που είναι αποθηκευμένο το `src`. Στο πεδίο `len` βρίσκεται το μήκος του `src`. Γνωρίζοντας αυτές τις δυο πληροφορίες αντιγράφουμε και το `src` από το user space του VM. Επίσης αντιγράφουμε και το iv που βρίσκεται στο `crypto_op.iv`.

4.2.5. /* AFTER SGS ARE READY FROM BACKEND */

```
    switch(cmd) {
    case (CIOCGSESSION):
        debug("CIOCGSESSION RET");
        if (unlikely(copy_to_user((struct session_op*)arg, &seop, sizeof(struct
session_op))))
            return -EFAULT;
        kfree(key);

        break;

    case (CIOCFSESSION):
        debug("CIOCFSESSION RET");

        break;

    case (CIOCCRYPT):
        debug("CIOCCRYPT RET");
        if (unlikely(copy_to_user((struct crypt_op*)arg, &crop, sizeof(struct
crypt_op))))
            return -EFAULT;
        if (unlikely(copy_to_user(crop.dst, dst, crop.len*sizeof(char))))
            return -EFAULT;
        kfree(src);
        kfree(dst);
        kfree(iv);

        break;
    }
```

Μόλις το back-end τελειώσει με την επεξεργασία των scatterlists, θα πρέπει να αντιγράψουμε στο userspace, στη διεύθυνση arg, το struct session_op, όπως έχει διαμορφωθεί, μετά και από το `ioctl CIOCGSESSION` και αντίστοιχα το crypt_op όπως έχει διαμορφωθεί, μετά και από το `ioctl CIOCCRYPT`. Αυτό επιτυγχάνεται μέσω της `copy_to_user`, η οποία κάνει την ακριβώς αντίστροφη λειτουργία της `copy_from_user`.

4.3. Περιγραφή υλοποίησης των επιμέρους τμημάτων του παραπάνω κώδικα του back end (qemu)

Πρόκειται για το κομμάτι της εφαρμογής που βρίσκεται στο user space του host. Φροντίζει να επεξεργαστεί τις scatterlists από το front end, το οποίο βρίσκεται στο kernel space του VM. Πιο συγκεκριμένα, πραγματοποιεί κλήσεις συστήματος στο host μηχάνημα, εκ μέρους του πυρήνα του VM. Μόλις οι κλήσεις αυτές πραγματοποιηθούν

αναλαμβάνει να τροφοδοτήσει με τα αποτελέσματά τους το front end, το οποίο με τη σειρά του τα επεξεργάζεται καταλλήλως.

```
VirtQueueElement elem;
unsigned int *syscall_type;
int *host_fd;

DEBUG_IN();

if (!virtqueue_pop(vq, &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) {
//...//
}
virtqueue_push(vq, &elem, 0);
virtio_notify(vdev, vq);
```

Στα πλαίσια της χρήσης του VirtIO, το back end επικοινωνεί με το front end μέσω μιας VirtQueue. Πιο συγκεκριμένα, η VirtQueue αυτή περιέχει ένα αντικείμενο VirtQueueElement, το οποίο περιλαμβάνει όλες τις scatterlists που ανταλλάσσονται κάθε στιγμή μεταξύ front end και back end. Συνεπώς το back end θα πρέπει να κάνει pop το αντικείμενο αυτό και στη συνέχεια να κάνει manipulate τις scatterlists που περιέχονται εντός του και οι οποίες έχουν αποσταλεί από το front end. Σε αυτό το σημείο να αναφέρουμε ότι δηλώνοντας τις μεταβλητές του front end ως stack allocated αντί heap allocated ότι στο pop «έσκαγε» το back end.

Η πρώτη scatterlist κάθε φορά είναι αυτή που αναφέρεται στο syscalltype και βάσει της τιμής του πραγματοποιείται μία από τις λειτουργίες `open`, `close`, `ioctl`. Εν συνεχεία και αφού ολοκληρωθεί η επεξεργασία των scatterlists, το back end θα πρέπει να κάνει push το VirtQueueElement στην VirtQueue, προκειμένου να το βρει το front end. Επίσης θα πρέπει να ειδοποιήσει το front προκειμένου το αποτέλεσμα της επεξεργασίας των scatterlists να περάσει σε αυτό. Αυτό πραγματοποιείται μέσω της κλήσης `virtio_notify`. Μόλις εκτελεστεί το notify, στο front-end η τιμή της `virtqueue_get_buf(crdev->vq, &len)` θα πάψει να είναι `NULL` και η εκτέλεση θα ξεμπλοκάρει από το while loop.

4.3.1. Received Scatterlist for System Call Open

```
host_fd = elem.in_sg[0].iov_base;
*host_fd = open(CRYPTODEV_FILENAME, O_RDWR);
```

Στην περίπτωση του system call open το front end έχει στείλει μία scatterlist προς εγγραφή, στην οποία θα πρέπει να γυρίσουμε το αποτέλεσμα του system call open του `/dev/crypto`. Εκεί λοιπόν θα πάμε και θα γράψουμε το αποτέλεσμα της open. Στο σημείο αυτό αξίζει να αναφερθεί πως δεν γίνεται κάποιος έλεγχος σφάλματος του open που πραγματοποιείται στο back-end. Αυτό γιατί τα errors θα πρέπει να ελέγχονται στο πρόγραμμα του user space του VM. Σε περίπτωση που γινόταν έλεγχος λάθους στο back-end, πιθανό λάθος θα οδηγούσε σε τερματισμό του qemu.

4.3.2. Received Scatterlist for System Call Close

```
host_fd = elem.out_sg[1].iov_base;
close(*host_fd);
```

Εδώ τα πράγματα είναι πολύ απλά. Το front end έχει στείλει μία scatterlist προς ανάγνωση. Από εκεί λαμβάνουμε τον file descriptor του device του host μηχανήματος, στο οποίο θα κάνουμε την κλήση συστήματος close της συσκευής `/dev/crypto`.

4.3.3. Received Scatterlist for System Call Ioctl

Κοινός κώδικας και των τριών ioctl υποπεριπτώσεων.

```
unsigned int *ioctl_cmd;
long *host_return_val;

host_fd = elem.out_sg[1].iov_base;
ioctl_cmd = elem.out_sg[2].iov_base;
```

4.3.3.1. CIOCGSESSION

```
unsigned char *session_key;
unsigned char *temp1;
struct session_op *session_op;
//read from here
session_key = elem.out_sg[3].iov_base;
session_op = elem.in_sg[0].iov_base;
//write here
host_return_val = elem.in_sg[1].iov_base;
//we save the key we recieved so that the frondend gets the right
address
temp1 = session_op->key;
session_op->key = session_key;
//syscall
*host_return_val = ioctl(*host_fd, CIOCGSESSION, session_op);
session_op->key = temp1;
```

Σε αυτή την περίπτωση, η `ioctl` command είναι η `CIOCGSESSION` για τη δημιουργία ενός session. Ο front end μας έχει αποστείλει επιπλέον από τα 2 scatterlists για εγγραφή και για ανάγνωση το session_key για την κρυπτογράφηση , το session_op και τη διεύθυνση επιστροφής `host_return_val`. Συμπληρώνουμε τα ορίσματα της `ioctl()` και εκτελούμε την κλήση συστήματος. Επίσης, αποθηκεύουμε τη διεύθυνση του key που βρίσκεται στο userspace του VM, πριν την εκτέλεση της `ioctl`, και την αντικαθιστούμε με τη διεύθυνση `session_key` που λάβαμε από τη scatterlist , ενώ στο τέλος επαναφέρουμε την αρχική διεύθυνση. Αυτό το κάνουμε έτσι ώστε στο fronend κομμάτι να επιστραφεί στο user η σωστή διεύθυνση του κλειδιού στο user space του VM. Στο τέλος, έχουμε γράψει το `session_op` καθώς και την τιμή που επέστρεψε η κλήση `ioctl()`.

4.3.3.2. CIOCFSESSION

```
uint32_t *ses_id;
//read from here
ses_id = elem.out_sg[3].iov_base;
//write here
host_return_val = elem.in_sg[0].iov_base;
// syscall
*host_return_val = ioctl(*host_fd, CIOCFSESSION, ses_id);
```

Στην περίπτωση αυτή το `ioctl` command που ζητείται είναι ο τερματισμός ενός session, `CIOCFSESSION`. Όπως και προηγουμένως χρησιμοποιούμε 2 scatterlist για ανάγνωση και για εγγραφή. Από αυτές παίρνουμε το `session_id` του session που θέλουμε να τερματίσουμε και τη διεύθυνση επιστροφής. Στην τελευταία επιστρέφουμε το αποτέλεσμα της κλήσης συστήματος.

4.3.3.3. CIOCCRYPT

```
struct crypt_op crypt_op;

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

host_return_val = elem.in_sg[1].iov_base;
*host_return_val = ioctl(*host_fd, CIOCCRYPT, &crypt_op);
```

Τελευταία πιθανή κλήση συστήματος που μπορεί να ζητηθεί είναι η `CIOCCRYPT` με την οποία ζητείται η κρυπτογράφηση ενός μυνήματος. Σε αυτή την περίπτωση από τις 2 scattrelists θα πάρουμε: το `struct crypt_op` καθώς και τις τιμές των εξής πεδίων τους `crypt_op.src, crypt_op.iv`, τη διεύθυνση επιστροφής, ενώ θα γράψουμε στο `crypt_op.dst`, που είναι και το αποτέλεσμα του `CIOCCRYPT`. Γεμίζουμε τα αντίστοιχα πεδία του struct γιατί πρέπει να αλλάξουν για την κλήση συστήματος. Μετά την εκτέλεση της κλήσης συστήματος θα γίνει επιστροφή στο `host_return_val` και το `dst` θα περιέχει το αποτέλεσμα της κρυπτογράφησης.