Εργαστήριο Λειτουργικών Συστημάτων

3η εργαστηριακή άσκηση
Κρυπτογραφική συσκευή VirtIO για QEMU-KVM
Αρβανίτης Χρήστος 03114622
Μπαγάκης Εμμανουήλ 03114157

Ζητούμενο 1: Εργαλείο chat πάνω από TCP/IP sockets

Στο συγκεκριμένο ζητούμενο απαιτείται η δημιουργία ενός chat προγράμματος για την επικοινωνία δύο συστημάτων. Η επικοινωνία βασίζεται στην χρήση sockets και συγκεκριμένα TCP sockets, σε μοντέλο client-server. Αρχικά αμφότεροι server και client δημιουργούν ένα socket με την κλήση της socket().

```
if ((sd = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
        perror("socket");
        exit(1);
       }
```

Ο server με την bind() δεσμεύει το socket με μια συγκεκριμένη διεύθυνση, ενώ με τη listen() ορίζει τις παραμέτρους που χρειάζονται για να μπορεί να δεχθεί το socket συνδέσεις.

```
 if (bind(sd, (struct sockaddr *)&sa, sizeof(sa)) < 0) {
        perror("bind");
        exit(1);
    }

  if (listen(sd, TCP_BACKLOG) < 0) {
        perror("listen");
        exit(1);
    }
```

Στον server ο client συνδέεται με την connect() σε συγκεκριμένη ip διεύθυνση (εκείνη του server, για μας εδώ η localhost) και θύρα.

```
if (connect(sd, (struct sockaddr *) &sa, sizeof(sa)) < 0) {
        perror("connect");
        exit(1);
```

Αυτά τα δύο χαρακτηριστικά του sa struct συμπληρώνονται με τις ακόλουθες κλήσεις:

```
 /* 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);
```

Με την accept() η οποία αναμένει έως ότου βρεθεί σύνδεση, ο server δημιουργεί ένα socket για επικοινωνία με τον client. Στη συνέχεια αμφότεροι επικοινωνούν αμφίδρομα με κλήσεις read() και write() έως ότου αποχωρήσει ένας από τους δύο από την σύνοδο.

```
if ((newsd = accept(sd, (struct sockaddr *)&sa, &len)) < 0) {
         perror("accept");
         exit(1);
    }
```

Η αμφίδρομη επικοινωνία επιτυγχάνεται με την χρήση της select η οποία παρακολουθεί τους write file descriptors και τους read file descriptors του καθενός (κάθε client και κάθε server έχει ξεχωριστή κλήση της select). Έτσι όταν ένας από αυτούς είναι “ready” για κάποια I/O λειτουργία, δηλαδή έχει δεδομένα προς επεξεργασία/ανάγνωση, η select επιστρέφει και ανάλογα με το fd το οποίο είναι έτοιμο (ανάγνωσης ή εγγραφής) λαμβάνουμε ή αποστέλλουμε δεδομένα μέσω των αντίστοιχων buffers. Οι read και write ομάδες descriptor χτίζονται από εμάς τους ίδιους χρησιμοποιώντας την build_fd_sets().

```
int activity = select(maxfd + 1, &read_fds, &write_fds,&except_fds, NULL);
```

Σε κάθε περίπτωση όταν αποχωρεί ο πελάτης με την close(), ο server κλείνει την τρέχουσα σύνοδο (close()) και ξανακαλείται η accept() η οποία αναμένει τον επόμενο προς σύνδεση πελάτη.

```
  if (close(newsd) < 0)
         perror("close");
```

Παράλληλα, πρέπει να κλείσει και το socket του server με κλήση της shutdown().

Ο κώδικας του συγκεκριμένου ερωτήματος παρατίθεται στη συνέχεια:

Socket Client

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

```c
#include <errno.h>
#include <stdio.h>
#include <signal.h>
#include <sys/select.h>
#include <netinet/in.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <errno.h>
#include <ctype.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>
#include <netdb.h>
#include <sys/time.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include "socket-common.h"
#include <crypto/cryptodev.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>

#define DATA_SIZE     256
#define BLOCK_SIZE    16
#define KEY_SIZE  16  /* AES128 */

int sd_global;
/* Convert a buffer to uppercase */
void toupper_buf(char *buf, size_t n)
{
    size_t i;

    for (i = 0; i < n; i++)
        buf[i] = toupper(buf[i]);
}

void handle_sigint(int sig)
{
    write(sd_global,"KILL_CONNECTION\n",sizeof("KILL_CONNECTION\n"));
    if (close(sd_global) < 0)
        perror("close");
    exit(1);
}

/* Insist until all of the data has been written */
ssize_t insist_write(int fd, const void *buf, size_t cnt)
```

```
{
    ssize_t ret;
    size_t orig_cnt = cnt;

    while (cnt > 0) {
        ret = write(fd, buf, cnt);
        if (ret < 0)
                return ret;
        buf += ret;
        cnt -= ret;
    }

    return orig_cnt;
}

int build_fd_sets(int socket, fd_set *read_fds, fd_set *write_fds,fd_set *except_fds)
{
    FD_ZERO(read_fds);
    FD_SET(STDIN_FILENO, read_fds);
    FD_SET(socket, read_fds);

    FD_ZERO(write_fds);
    // there is smth to send, set up write_fd for server socket
    FD_SET(socket, write_fds);
    FD_ZERO(except_fds);
    FD_SET(STDIN_FILENO, except_fds);
    FD_SET(socket, except_fds);
    return 0;
}

/* Insist until all of the data has been read */
ssize_t insist_read(int fd, void *buf, size_t cnt)
{
    ssize_t ret;
    size_t orig_cnt = cnt;

    while (cnt > 0) {
        ret = read(fd, buf, cnt);
        if (ret < 0)
                return ret;
        buf += ret;
        cnt -= ret;
    }

    return orig_cnt;
}

static int fill_urandom_buf(unsigned char *buf, size_t cnt)
{
    int crypto_fd;
```

```
    int ret = -1;

    crypto_fd = open("/dev/urandom", O_RDONLY);
    if (crypto_fd < 0)
        return crypto_fd;

    ret = insist_read(crypto_fd, buf, cnt);
    close(crypto_fd);

    return ret;
}

int main(int argc, char *argv[])
{
    fd_set read_fds;
    fd_set write_fds;
    fd_set except_fds;

    int cfd;
    struct sockaddr_in sa;
    struct session_op sess;
    int sd, port,crypto_fd;
    ssize_t n;
    char buf[DATA_SIZE],buf2[DATA_SIZE];
    char *hostname;
    struct hostent *hp;

    /* Argument check */
    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 */

    /* Set nonblock for stdin. */
    int flag = fcntl(STDIN_FILENO, F_GETFL, 0);
    flag |= O_NONBLOCK;
    fcntl(STDIN_FILENO, F_SETFL, flag);

    memset(buf2, 0, DATA_SIZE);
    memset(buf, 0, DATA_SIZE);
    cfd = open("/dev/crypto",O_RDONLY); //Open crypto device and get fd

    /* 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);
    }
    sd_global = sd;
    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");

    int maxfd = sd; //max file descriptor to check duriong select

    for (;;) {
        build_fd_sets(sd,&read_fds,&write_fds,&except_fds);

        int activity = select(maxfd + 1, &read_fds, &write_fds,&except_fds, NULL);

        switch (activity) {
               case -1:
                      perror("select()");

               case 0:
                      // you should never get here
                      printf("select() returns 0.\n");

               default:
                      /* All fd_set's should be checked. */
                      if (FD_ISSET(STDIN_FILENO, &read_fds)) { //we got data to stdin time
to send them
                             printf("Sending data to socket.\n");
                             memset(buf, 0, DATA_SIZE);
                             n = read(STDIN_FILENO, buf, sizeof(buf));
                             if(n==0){
                                    fprintf(stderr, "Peer went away\n");
                                           if (shutdown(sd, SHUT_WR) < 0) {
                                                  perror("shutdown");
                                                  exit(1);}
```

```
                                        if (close(sd) < 0)
                                                perror("close");
                                        exit(1);
                                }

                                if (insist_write(sd, buf, DATA_SIZE) != DATA_SIZE) {
                                        perror("write to remote peer failed");
                                        break;
                                }
                                continue;
                        }

                        if (FD_ISSET(STDIN_FILENO, &except_fds)) {
                                printf("except_fds for stdin.\n");
                        }

                        if (FD_ISSET(sd, &read_fds)) { //we got data on socket. Decrypt them
and give them to stdout
                                memset(buf2, 0, DATA_SIZE);
                                printf("Getting data from sd.\n");
                                n = read(sd, buf2, DATA_SIZE);
                                if (n <= 0) {
                                        if (n < 0)
                                                perror("read from remote peer failed");
                                        else
                                                fprintf(stderr, "Peer went away\n");
                                        close(sd);
                                        exit(1);
                                        break;
                                }

                                write(STDOUT_FILENO,buf2,DATA_SIZE); //Print to stdout

                        }

                        if (FD_ISSET(sd, &except_fds)) {
                                printf("except_fds to socket\n");
                        }
                }
        }
        if (close(sd) < 0)
                perror("close");
        exit(1);
        //sig handler for closing sd
        /* This will never happen */
        return 1;
}
```

Socket Server

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


#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <sys/select.h>
#include <netinet/in.h>
#include <errno.h>
#include <ctype.h>
#include <string.h>
#include <netdb.h>
#include <sys/time.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include "socket-common.h"
#include <crypto/cryptodev.h>
#include <fcntl.h>
#include <stdlib.h>
#include <sys/ioctl.h>
#include <sys/stat.h>

#define DATA_SIZE     256
#define BLOCK_SIZE    16
#define KEY_SIZE 16  /* AES128 */

int newsd_global;
/* Convert a buffer to upercase */
void toupper_buf(char *buf, size_t n)
{
    size_t i;

    for (i = 0; i < n; i++)
        buf[i] = toupper(buf[i]);
}

void handle_sigint(int sig)
{
```

```
    write(newsd_global,"KILL_CONNECTION\n",sizeof("KILL_CONNECTION\n"));
    if (close(newsd_global) < 0)
        perror("close");
    exit(1);
}

/* Insist until all of the data has been written */
ssize_t insist_write(int fd, const void *buf, size_t cnt)
{
    ssize_t ret;
    size_t orig_cnt = cnt;

    while (cnt > 0) {
        ret = write(fd, buf, cnt);
        if (ret < 0)
                return ret;
        buf += ret;
        cnt -= ret;
    }

    return orig_cnt;
}

int build_fd_sets(int socket, fd_set *read_fds, fd_set *write_fds,fd_set *except_fds)
{
    FD_ZERO(read_fds);
    FD_SET(STDIN_FILENO, read_fds);
    FD_SET(socket, read_fds);

    FD_ZERO(write_fds);
    // there is smth to send, set up write_fd for server socket
    FD_SET(socket, write_fds);
    FD_ZERO(except_fds);
    FD_SET(STDIN_FILENO, except_fds);
    FD_SET(socket, except_fds);
    return 0;
}
/* Insist until all of the data has been read */
ssize_t insist_read(int fd, void *buf, size_t cnt)
{
    ssize_t ret;
    size_t orig_cnt = cnt;

    while (cnt > 0) {
        ret = read(fd, buf, cnt);
        if (ret < 0)
                return ret;
        buf += ret;
        cnt -= ret;
    }
```

```
    return orig_cnt;
}

static int fill_urandom_buf(unsigned char *buf, size_t cnt)
{
    int crypto_fd;
    int ret = -1;

    crypto_fd = open("/dev/urandom", O_RDONLY);
    if (crypto_fd < 0)
        return crypto_fd;

    ret = insist_read(crypto_fd, buf, cnt);
    close(crypto_fd);

    return ret;
}

int main(void)
{
    char buf[DATA_SIZE],buf2[DATA_SIZE];
    char addrstr[INET_ADDRSTRLEN];
    int sd, newsd,cfd;
    ssize_t n;
    socklen_t len;
    struct sockaddr_in sa;



    /* Set nonblock for stdin. */
    int flag = fcntl(STDIN_FILENO, F_GETFL, 0);
    flag |= O_NONBLOCK;
    fcntl(STDIN_FILENO, F_SETFL, flag);

    fd_set read_fds;
    fd_set write_fds;
    fd_set except_fds;

    cfd = open("/dev/crypto",O_RDONLY); //Open crypto device and get fd


    memset(buf2, 0, DATA_SIZE);
    memset(buf, 0, DATA_SIZE);

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

    //signal(SIGINT, handle_sigint);
```

```
/* 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 */

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);
}
newsd_global = newsd;
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));

int maxfd = newsd;

for (;;) {
    //something is sketchy with those memsets
    //memset(buf, 0, DATA_SIZE);
    memset(buf2, 0, DATA_SIZE);
    build_fd_sets(newsd,&read_fds,&write_fds,&except_fds);
```

```
int activity = select(maxfd + 1, &read_fds, &write_fds,&except_fds, NULL);

switch (activity) {
	case -1:
		perror("select()");

	case 0:
		// you should never get here
		printf("select() returns 0.\n");

	default:
		/* All fd_set's should be checked. */
		if (FD_ISSET(STDIN_FILENO, &read_fds)) {
			printf("Sending data to socket.\n");
			memset(buf, 0, DATA_SIZE);
			n = read(STDIN_FILENO, buf, sizeof(buf));
			if(n==0){
				fprintf(stderr, "Peer went away\n");
					if (shutdown(newsd, SHUT_WR) < 0) {
						perror("shutdown");
						exit(1);}
						if (close(sd) < 0)
							perror("close");
						exit(1);
			}
			/* Be careful with buffer overruns, ensure NUL-termination */
			//buf[sizeof(buf) - 1] = '\0';

			if (insist_write(newsd,buf, DATA_SIZE) != DATA_SIZE) {
				perror("write to remote peer failed");
				break;
			}
			continue;
		}

		if (FD_ISSET(STDIN_FILENO, &except_fds)) {
			printf("except_fds for stdin.\n");
		}

		if (FD_ISSET(newsd, &read_fds)) {
			printf("Getting data from sd.\n");
			memset(buf2, 0, DATA_SIZE);
			//n = read(newsd, buf, sizeof(buf));
			n = read(newsd, buf2, DATA_SIZE);

			if (n <= 0) {
				if (n < 0)
					perror("read from remote peer failed");
				else{
					fprintf(stderr, "Peer went away\n");
```

```
                                        if (close(newsd) < 0)
                                                perror("close");
                                        exit(1);
                                        if (close(sd) < 0)
                                                perror("close");
                                        exit(1);
                                }
                                /*
                                if (close(newsd) < 0)
                                        perror("close");

                                if (close(sd) < 0)
                                        perror("close");
                                        */
                                exit(1);
                                break;
                        }

                        //              printf("%s\n","Something happend" );
                        write(STDOUT_FILENO,buf2,DATA_SIZE); //Print to stdout
                        memset(buf, 0, DATA_SIZE);
                }



                if (FD_ISSET(sd, &write_fds)) {
                        printf("write to socket.\n");
                }

                if (FD_ISSET(sd, &except_fds)) {
                        printf("except_fds to socket\n");
                }
        }

    }
    if (close(newsd) < 0)
        perror("close");

    if (close(sd) < 0)
        perror("close");
    /* This will never happen */
    return 1;
}
```

Ζητούμενο 2: Κρυπτογραφημένο chat πάνω από TCP/IP

Αυτό το ζητούμενο διαφοροποιείται από το προηγούμενο ως προς το ότι πλέον δεν ανταλλάσσουμε απλά μηνύματα μέσω του TCP socket που δημιουργήσαμε προηγουμένως, αλλά χρησιμοποιούμε τη κρυπτογραφική συσκευή /dev/crypto ώστε να κρυπτογραφήσουμε κάθε μήνυμα πριν την αποστολή και να απο-κρυπτογραφήσουμε αντίστοιχα κατά τη λήψη του. Προς αυτή τη κατεύθυνση, χρησιμοποιούμε τις κλήσεις open() και ioctl() στη συγκεκριμένη συσκευή. Αρχικά ανοίγουμε το αρχείο /dev/crypto με τη σημαία RD_ONLY. Στη συνέχεια, αρχικοποιούμε μια σύνοδο μέσω της κλήσης ioctl():

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

Στη δομή session επιστρέφονται οι πληροφορίες της συνόδου με τη συσκευή. Αφού έχουμε λάβει το session, χρησιμοποιούμε τη δομή crypt_op cryp, όπως απαιτείται από το API ώστε να συμπληρώσουμε απαραίτητες πληροφορίες για κάθε λειτουργία κρυπτογράφησης/αποκρυπτογράφησης η οποία θα συντελεστεί στο πρόγραμμά μας. Συγκεκριμένα:

```
    cryp.ses = sess.ses;
    cryp.len = sizeof(data.in); //THis is somewhat fixed to DATA_SIZE for now
    cryp.src = (void *)data.in;
    cryp.dst = data.encrypted;
    cryp.iv = (void *)data.iv;
```

Το μήκος κάθε string το οποίο επεξεργάζεται η κρυπτογραφική συσκευή ορίζεται fixed στο μέγεθος του data.in πεδίου το οποίο θα αναλύσουμε στη συνέχεια. Το iv πεδίο του data χρησιμοποιείται στο πρώτο στάδιο της κρυπτογράφησης και όπως θα δούμε παρακάτω αρχικοποιείται σε μηδενικά χάριν ευκολίας στη συγκεκριμένη άσκηση, με τη πρόθεση να μεταφέρεται με κάποιον τρόπο ως prefix στα μηνύματα ώστε να γίνει randomized για μεγαλύτερη ασφάλεια. Επίσης, το κλειδί το οποίο είναι γνωστό και στις δύο πλευρές, χρησιμοποιείται τόσο στην κρυπτογράφηση όσο και στην κρυπτογράφηση των μηνυμάτων και από τις δύο πλευρές. Αποφασίσαμε για λόγους ευκολίας να το αρχικοποιήσουμε σε μηδενική τιμή (αξίζει να αναφέρουμε ότι η μέθοδος κρυπτογράφησης AES-256 ακόμη και με μηδενικό κλειδί και iv παράγει κρυπτοκείμενο διαφορετικό του αρχικού). Ας δούμε αναλυτικότερα τη δομή data που χρησιμοποιήσαμε:

```
    struct {
        unsigned char in[DATA_SIZE],
                      encrypted[DATA_SIZE],
                      decrypted[DATA_SIZE],
                      iv[BLOCK_SIZE],
                      key[KEY_SIZE];
```

```
} data; //crypto structure
```

Σε αυτή τη δομή, τα πεδία in, encrypted και decrypted αποτελούν πεδία που δέχονται την είσοδο και φιλοξενούν την έξοδο κάθε ενέργειας encrypt/decrypt. Όλα γίνονται fixed size στο DATA_SIZE macro. Συνεπώς τα μηνύματά μας μεταδίδονται σε “κβάντα” των 256 χαρκτήρων, ενώ αφού στη συνέχεια αρχικοποιήσουμε κάθε πεδίο χαρακτήρων στο κενό string, οι υπόλοιποι κενοί χαρακτήρες απλά δεν θα τυπωθούν. Ο συγκεκριμένος περιορισμός στο μήκος των συμβολοσειρών δεν αποτελεί πρόβλημα ούτε στη περίπτωση που έχουμε μεγαλύτερο των 256 χαρακτήρων μήνυμα, αφού στο πρώτο read θα διαβαστούν οι πρώτοι χαρακτήρες και στη συνέχεια, στην επόμενη επανάληψη οι υπόλοιποι, διατηρώντας πάντα το σταθερό μέγεθος μηνύματος. Οι αρχικοποιήσεις κάθε πεδίου γίνονται με τη memset όπως ακολούθως:

```
memset(&sess, 0, sizeof(sess));
memset(&cryp, 0, sizeof(cryp));
memset(&data, 0, sizeof(data)); //set all values to zero (even iv and key values)

memset(buf2, 0, DATA_SIZE);
memset(buf, 0, DATA_SIZE);
```

Παραπάνω αρχικοποιούμε και δύο buffers οι οποίοι χρησιμοποιούνται για το encrypt/decrypt. Συγκεκριμένα, σε περίπτωση που υπάρχουν διαθέσιμα δεδομένα για αποστολή, αυτά θα πρέπει να κρυπτογραφηθούν. Αρχικά, διαβάζουμε από το αρχείο που αντιστοιχεί στο STDIN και αν επιστραφεί 0 κλείνουμε μέσω του shutdown το socket καθως και το ίδιο το αρχείο sd. Αντίθετα, αν υπάρχουν αξιοποιήσιμα δεδομένα τα αντιγράφουμε από τον buffer που αντιστοιχεί στα δεδομένα του stdin στον buf. Ορίζουμε το operation ως COP_ENCRYPT δηλαδή κρυπτογράφηση και στη συνέχεια αντιγράφουμε στη δομή cryp τα δεδομένα προς επεξεργασία. Καλώντας την κατάλληλη ioctl() με το flag CIOCRYPT κρυπτογραφούμε τα δεδομένα οπότε και είμαστε έτοιμοι να τα λάβουμε από το destination field. Δεν αμελούμε να μηδενίσουμε τον buffer buf για να είναι έτοιμος για την επόμενη επανάληψη. Χρησιμοποιώντας την συνάρτηση insist_write ώστε να είμαστε βέβαιοι πως τα δεδομένα μας θα γραφούν ακέραια στο socket, στέλνουμε τα δεδομένα στο άλλο άκρο του TCP connection.

```
if (FD_ISSET(STDIN_FILENO, &read_fds)) { //we got data to stdin time to send them
                            printf("Sending data to socket.\n");
                            n = read(STDIN_FILENO, buf, sizeof(buf));
                            if(n==0){
                                    fprintf(stderr, "Peer went away\n");
                                            if (shutdown(sd, SHUT_WR) < 0) {
                                                    perror("shutdown");
                                                    exit(1);}
                                                    if (close(sd) < 0)
                                                            perror("close");
                                                if (ioctl(cfd,CIOCFSESSION, &sess)) {
                                                     perror("ioctl(CIOCFSESSION)");
                                                     return 1;
```

```
                        }

                          exit(1);
                }

                cryp.len = DATA_SIZE;
                cryp.src = buf;
                cryp.op = COP_ENCRYPT;
                cryp.dst = data.encrypted;
                if (ioctl(cfd, CIOCCRYPT, &cryp)) {
                        perror("ioctl(CIOCCRYPT)");
                        return 1;
                }
                memset(buf, 0, DATA_SIZE);

                if (insist_write(sd, cryp.dst, DATA_SIZE) != DATA_SIZE) {
                        perror("write to remote peer failed");
                        break;
                }
                continue;
        }
```

Στη περίπτωση που υπάρχουν δεδομένα προς ανάγνωση από το απομακρυσμένο άκρο, δηλαδή το socket, θα γίνει raise το κατάλληλο flag από τη select οπότε και θα πρέπει να αποκρυπτογραφήσουμε τα δεδομένα προτού τα τυπώσουμε στο stdout. Αναλυτικότερα:

```
        if (FD_ISSET(sd, &read_fds)) { //we got data on socket. Decrypt them
and give them to stdout
                printf("Getting data from sd.\n");
                n = read(sd, buf2, DATA_SIZE);
                if (n <= 0) {
                        if (n < 0)
                                perror("read from remote peer failed");
                        else
                                fprintf(stderr, "Peer went away\n");

                        close(sd);
                        exit(1);
                        break;
                }
                /*
                 * Decrypt data.encrypted to data.decrypted
                 */

                //strcpy(data.in,buf2); //get data to decrytpion input
                cryp.src = buf2;
                cryp.dst = buf;
                cryp.op = COP_DECRYPT;
```

```
                if (ioctl(cfd, CIOCCRYPT, &cryp)) {
                        perror("ioctl(CIOCCRYPT)");
                        return 1;
                }
                memset(buf2, 0, DATA_SIZE);

                write(STDOUT_FILENO,buf,DATA_SIZE); //Print to stdout
                memset(buf, 0, DATA_SIZE);
        }
```

Αν τα δεδομένα είναι κενά γνωρίζουμε πως όπως και πριν πρέπει να κλείσουμε τη σύνδεση ενώ επιπροσθέτως θα κλείσουμε και τη σύνοδο με τη κρυπτογραφική συσκευή. Διαβάζουμε τα encrypted δεδομένα απο το socket και τα αντιγράφουμε στο buf2 το οποίο θα αποτελέσει είσοδο για την διαδικασία COP_DECRYPT, ενώ έξοδο αποτελεί ο buf. Στη συνέχεια καλούμε την ioctl(), και αφού τελεσθεί η κρυπτογράφηση αντιγράφουμε τα δεδομένα με insist_write() στο stdout προς ανάγνωση. Σε όλη αυτή τη διαδικασία δεν αμελούμε να μηδενίσουμε και τους δύο buffers για την επόμενη επανάληψη που θα χρειασθούν.

Ο κώδικας του συγκεκριμένου ερωτήματος παρατίθεται στη συνέχεια:

Socket Client

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


#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <sys/select.h>
#include <netinet/in.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <ctype.h>
#include <string.h>
#include <netdb.h>
#include <sys/time.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
```

```c
#include <netinet/in.h>
#include "socket-common.h"
#include <crypto/cryptodev.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>

#define DATA_SIZE     256
#define BLOCK_SIZE    16
#define KEY_SIZE  16  /* AES128 */

int sd_global;
/* Convert a buffer to upercase */
void toupper_buf(char *buf, size_t n)
{
    size_t i;

    for (i = 0; i < n; i++)
        buf[i] = toupper(buf[i]);
}

void handle_sigint(int sig)
{
    write(sd_global,"KILL_CONNECTION\n",sizeof("KILL_CONNECTION\n"));
    if (close(sd_global) < 0)
        perror("close");
    exit(1);
}

/* Insist until all of the data has been written */
ssize_t insist_write(int fd, const void *buf, size_t cnt)
{
    ssize_t ret;
    size_t orig_cnt = cnt;

    while (cnt > 0) {
        ret = write(fd, buf, cnt);
        if (ret < 0)
                return ret;
        buf += ret;
        cnt -= ret;
    }

    return orig_cnt;
}

int build_fd_sets(int socket, fd_set *read_fds, fd_set *write_fds,fd_set *except_fds)
{
```

```
    FD_ZERO(read_fds);
    FD_SET(STDIN_FILENO, read_fds);
    FD_SET(socket, read_fds);

    FD_ZERO(write_fds);
    // there is smth to send, set up write_fd for server socket
    FD_SET(socket, write_fds);
    FD_ZERO(except_fds);
    FD_SET(STDIN_FILENO, except_fds);
    FD_SET(socket, except_fds);
    return 0;
}

/* Insist until all of the data has been read */
ssize_t insist_read(int fd, void *buf, size_t cnt)
{
    ssize_t ret;
    size_t orig_cnt = cnt;

    while (cnt > 0) {
        ret = read(fd, buf, cnt);
        if (ret < 0)
                return ret;
        buf += ret;
        cnt -= ret;
    }

    return orig_cnt;
}

static int fill_urandom_buf(unsigned char *buf, size_t cnt)
{
    int crypto_fd;
    int ret = -1;

    crypto_fd = open("/dev/urandom", O_RDONLY);
    if (crypto_fd < 0)
        return crypto_fd;

    ret = insist_read(crypto_fd, buf, cnt);
    close(crypto_fd);

    return ret;
}

int main(int argc, char *argv[])
{
    fd_set read_fds;
    fd_set write_fds;
    fd_set except_fds;
```

```
int cfd;
struct sockaddr_in sa;
struct session_op sess;
struct crypt_op cryp;
int sd, port,crypto_fd;
ssize_t n;
char buf[DATA_SIZE],buf2[DATA_SIZE];
char *hostname;
struct hostent *hp;
struct {
    unsigned char in[DATA_SIZE],
                  encrypted[DATA_SIZE],
                  decrypted[DATA_SIZE],
                  iv[BLOCK_SIZE],
                  key[KEY_SIZE];
} data; //crypto structure

/* Argument check */
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 */


/* Set nonblock for stdin. */
int flag = fcntl(STDIN_FILENO, F_GETFL, 0);
flag |= O_NONBLOCK;
fcntl(STDIN_FILENO, F_SETFL, flag);

/* Set and initialize all structures associated with crypto functions to 0 */
memset(&sess, 0, sizeof(sess));
memset(&cryp, 0, sizeof(cryp));
memset(&data, 0, sizeof(data)); //set all values to zero (even iv and key values)

memset(buf2, 0, DATA_SIZE);
memset(buf, 0, DATA_SIZE);
cfd = open("/dev/crypto",O_RDONLY); //Open crypto device and get fd

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

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

```
    }

    /* Set cryp struct fields */
    cryp.ses = sess.ses;
    cryp.len = sizeof(data.in); //THis is somewhat fixed to DATA_SIZE for now
    cryp.src = (void *)data.in;
    cryp.dst = data.encrypted;
    cryp.iv = (void *)data.iv;
    //cryp.op = COP_DECRYPT; //we set it after we know if we have to read from sd or write
to it

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

    /* Signal handler if Ctrl+C is pressed */
    //signal(SIGINT, handle_sigint);


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

    int maxfd = sd; //max file descriptor to check duriong select

    //fprintf(stderr, "Waiting for an incoming connection...\n");


    for (;;) {
        build_fd_sets(sd,&read_fds,&write_fds,&except_fds);
```

```
        int activity = select(maxfd + 1, &read_fds, &write_fds,&except_fds, NULL);

        switch (activity) {
                case -1:
                        perror("select()");

                case 0:
                        // you should never get here
                        printf("select() returns 0.\n");

                default:
                        /* All fd_set's should be checked. */
                        if (FD_ISSET(STDIN_FILENO, &read_fds)) { //we got data to stdin time
to send them
                                printf("Sending data to socket.\n");
                                n = read(STDIN_FILENO, buf, sizeof(buf));
                                if(n==0){
                                        fprintf(stderr, "Peer went away\n");
                                                if (shutdown(sd, SHUT_WR) < 0) {
                                                        perror("shutdown");
                                                        exit(1);}
                                                        if (close(sd) < 0)
                                                                perror("close");
                                                    if (ioctl(cfd,CIOCFSESSION, &sess)) {
                                                          perror("ioctl(CIOCFSESSION)");
                                                             return 1;
                                                 }

                                                        exit(1);
                                }

                                cryp.len = DATA_SIZE;
                                cryp.src = buf;
                                cryp.op = COP_ENCRYPT;
                                cryp.dst = data.encrypted;
                                if (ioctl(cfd, CIOCCRYPT, &cryp)) {
                                        perror("ioctl(CIOCCRYPT)");
                                        return 1;
                                }
                                memset(buf, 0, DATA_SIZE);

                                if (insist_write(sd, cryp.dst, DATA_SIZE) != DATA_SIZE) {
                                        perror("write to remote peer failed");
                                        break;
                                }
                                continue;
                        }

                        if (FD_ISSET(STDIN_FILENO, &except_fds)) {
```

```
                        printf("except_fds for stdin.\n");
                }

                if (FD_ISSET(sd, &read_fds)) { //we got data on socket. Decrypt them
and give them to stdout
                        printf("Getting data from sd.\n");
                        n = read(sd, buf2, DATA_SIZE);
                        if (n <= 0) {
                                if (n < 0)
                                        perror("read from remote peer failed");
                                else
                                        fprintf(stderr, "Peer went away\n");
                                close(sd);
                                exit(1);
                                break;
                        }
                        /*
                         * Decrypt data.encrypted to data.decrypted
                         */

                        //strcpy(data.in,buf2); //get data to decrytpion input
                        cryp.src = buf2;
                        cryp.dst = buf;
                        cryp.op = COP_DECRYPT;

                        if (ioctl(cfd, CIOCCRYPT, &cryp)) {
                                perror("ioctl(CIOCCRYPT)");
                                return 1;
                        }
                        memset(buf2, 0, DATA_SIZE);

                        write(STDOUT_FILENO,buf,DATA_SIZE); //Print to stdout
                        memset(buf, 0, DATA_SIZE);
                }

                if (FD_ISSET(sd, &except_fds)) {
                        printf("except_fds to socket\n");
                }
        }
    }
    if (close(sd) < 0)
        perror("close");
    exit(1);
    //sig handler for closing sd
    /* This will never happen */
    return 1;
}
```

Socket Server

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

#include <fcntl.h>
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <sys/select.h>
#include <netinet/in.h>
#include <errno.h>
#include <ctype.h>
#include <string.h>
#include <stdlib.h>
#include <netdb.h>
#include <sys/time.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include "socket-common.h"
#include <crypto/cryptodev.h>


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

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

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

#define DATA_SIZE     256
#define BLOCK_SIZE    16
#define KEY_SIZE 16  /* AES128 */

int newsd_global;
/* Convert a buffer to upercase */
void toupper_buf(char *buf, size_t n)
{
    size_t i;
```

```
    for (i = 0; i < n; i++)
        buf[i] = toupper(buf[i]);
}

void handle_sigint(int sig)
{
    write(newsd_global,"KILL_CONNECTION\n",sizeof("KILL_CONNECTION\n"));
    if (close(newsd_global) < 0)
        perror("close");
    exit(1);
}

/* Insist until all of the data has been written */
ssize_t insist_write(int fd, const void *buf, size_t cnt)
{
    ssize_t ret;
    size_t orig_cnt = cnt;

    while (cnt > 0) {
        ret = write(fd, buf, cnt);
        if (ret < 0)
                return ret;
        buf += ret;
        cnt -= ret;
    }

    return orig_cnt;
}

int build_fd_sets(int socket, fd_set *read_fds, fd_set *write_fds,fd_set *except_fds)
{
    FD_ZERO(read_fds);
    FD_SET(STDIN_FILENO, read_fds);
    FD_SET(socket, read_fds);

    FD_ZERO(write_fds);
    // there is smth to send, set up write_fd for server socket
    FD_SET(socket, write_fds);
    FD_ZERO(except_fds);
    FD_SET(STDIN_FILENO, except_fds);
    FD_SET(socket, except_fds);
    return 0;
}
/* Insist until all of the data has been read */
ssize_t insist_read(int fd, void *buf, size_t cnt)
{
    ssize_t ret;
    size_t orig_cnt = cnt;

    while (cnt > 0) {
```

```
        ret = read(fd, buf, cnt);
        if (ret < 0)
                return ret;
        buf += ret;
        cnt -= ret;
    }

    return orig_cnt;
}

static int fill_urandom_buf(unsigned char *buf, size_t cnt)
{
    int crypto_fd;
    int ret = -1;

    crypto_fd = open("/dev/urandom", O_RDONLY);
    if (crypto_fd < 0)
        return crypto_fd;

    ret = insist_read(crypto_fd, buf, cnt);
    close(crypto_fd);

    return ret;
}

int main(void)
{
    char buf[DATA_SIZE],buf2[DATA_SIZE];
    char addrstr[INET_ADDRSTRLEN];
    int sd, newsd,cfd;
    ssize_t n;
    socklen_t len;
    struct sockaddr_in sa;
    struct session_op sess;
    struct crypt_op cryp;
    struct {
        unsigned char in[DATA_SIZE],
        encrypted[DATA_SIZE],
        decrypted[DATA_SIZE],
        iv[BLOCK_SIZE],
        key[KEY_SIZE];
    } data;

    /* Set nonblock for stdin. */
    int flag = fcntl(STDIN_FILENO, F_GETFL, 0);
    flag |= O_NONBLOCK;
    fcntl(STDIN_FILENO, F_SETFL, flag);

    fd_set read_fds;
    fd_set write_fds;
```

```
fd_set except_fds;

cfd = open("/dev/cryptodev0",O_RDONLY); //Open crypto device and get fd


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

memset(buf2, 0, DATA_SIZE);
memset(buf, 0, DATA_SIZE);

sess.cipher = CRYPTO_AES_CBC;
sess.keylen = KEY_SIZE;
sess.key = data.key;

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

/*
 * set struct values for encrypt decrypt functions
 */
cryp.ses = sess.ses;
cryp.len = sizeof(data.in); //THis is somewhat fixed to DATA_SIZE for now
cryp.src = (void *)data.in;
cryp.dst = data.encrypted;
cryp.iv = (void *)data.iv;
//cryp.op = COP_ENCRYPT;

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

//signal(SIGINT, handle_sigint);

/* 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 */

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);
}
newsd_global = newsd;
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));

int maxfd = newsd;

for (;;) {
    //something is sketchy with those memsets
    //memset(buf, 0, DATA_SIZE);
    memset(buf2, 0, DATA_SIZE);
    build_fd_sets(newsd,&read_fds,&write_fds,&except_fds);

    int activity = select(maxfd + 1, &read_fds, &write_fds,&except_fds, NULL);

    switch (activity) {
            case -1:
                    perror("select()");

            case 0:
                    // you should never get here
                    printf("select() returns 0.\n");

            default:
                    /* All fd_set's should be checked. */
                    if (FD_ISSET(STDIN_FILENO, &read_fds)) {
                            printf("Sending data to socket.\n");
```

```
                        n = read(STDIN_FILENO, buf, sizeof(buf));
                        if(n==0){
                                fprintf(stderr, "Peer went away\n");
                                if (shutdown(sd, SHUT_WR) < 0) {
                                        perror("shutdown");
                                        exit(1);}
                                if (close(sd) < 0)
                                        perror("close");
                             if (ioctl(cfd,CIOCFSESSION, &sess)) {
                                 perror("ioctl(CIOCFSESSION)");
                                 return 1;
                             }
                                exit(1);
                        }

                        cryp.len = DATA_SIZE;
                        cryp.src = buf;
                        cryp.op = COP_ENCRYPT;
                        cryp.dst = data.encrypted;

                        if (ioctl(cfd, CIOCCRYPT, &cryp)) {
                                perror("ioctl(CIOCCRYPT)");
                                return 1;
                        }
                        memset(buf, 0, DATA_SIZE);
                        /* Be careful with buffer overruns, ensure NUL-termination */
                        //buf[sizeof(buf) - 1] = '\0';

                        if (insist_write(newsd, data.encrypted, DATA_SIZE) !=
DATA_SIZE) {
                                perror("write to remote peer failed");
                                break;
                        }
                        continue;
                }

                if (FD_ISSET(STDIN_FILENO, &except_fds)) {
                        printf("except_fds for stdin.\n");
                }

                if (FD_ISSET(newsd, &read_fds)) {
                        printf("Getting data from sd.\n");

                        //n = read(newsd, buf, sizeof(buf));
                        n = read(newsd, buf2, DATA_SIZE);

                        if (n <= 0) {
                                if (n < 0)
                                        perror("read from remote peer failed");
                                else
```

```
                                        fprintf(stderr, "Peer went away\n");
                                close(newsd);
                                exit(1);

                                break;
                        }

                        cryp.src = buf2;
                        cryp.dst = buf;
                        cryp.op = COP_DECRYPT;

                        if (ioctl(cfd, CIOCCRYPT, &cryp)) {
                                perror("ioctl(CIOCCRYPT)");
                                return 1;
                        }
                        memset(buf2, 0, DATA_SIZE);

                        //              printf("%s\n","Something happend" );
                        write(STDOUT_FILENO,buf,DATA_SIZE); //Print to stdout
                        memset(buf, 0, DATA_SIZE);
                }



                if (FD_ISSET(sd, &write_fds)) {
                        printf("write to socket.\n");
                }

                if (FD_ISSET(sd, &except_fds)) {
                        printf("except_fds to socket\n");
                }
        }

    }
    if (close(newsd) < 0)
        perror("close");

    if (close(sd) < 0)
        perror("close");
    /* This will never happen */
    return 1;
}
```

Ζητούμενο 3: Υλοποίηση συσκευής cryptodev με VirtIO

Σε αυτό το ζητούμενο, υλοποιούμε το σύστημα εκείνο το οποίο μας επιτρέπει με το ίδιο interface του cryptodev API να χρησιμοποιούμε εντός ενός VM όχι το virtual hardware /dev/crypto αλλά το αντίστοιχο native του host στον οποίο και εκτελείται το QEMU. Ο host λαμβάνει κλήσεις από το kernel space του guest οπότε και από το userspace του εκτελεί τις αντίστοιχες κλήσεις συστήματος πού είδαμε στο προηγούμενο ερώτημα. Από το παραπάνω φαίνεται πως ο οδηγός μας έχει δύο μέρη, το frontend μέρος που εκτελείται στα πλαίσια του driver στον guest και το backend το οποίο εκτελείται στα πλαίσια του userspace του host. Η επικοινωνία μεταξύ των δύο τμημάτων βασίζεται στο πρότυπο VirtIO. Χρησιμοποιούνται κυκλικοί buffers μέσω των οποίων μεταφέρουμε από το ένα άκρο στο άλλο τόσο την ωφέλιμη πληροφορία (περιεχόμενο μηνύματος) όσο και τον τύπο εντολής που θέλουμε το backend να εκτελέσει (δηλαδή ακεραίους που αντιστοιχούν σε συγκεκριμένα macros και ορίζονται στο κατάλληλο header file).

Στη συνέχεια περιγράφουμε συνοπτικά τα κύρια σημεία της υλοποίησής μας:

Περιγραφή frontend

Στο frontend μέρος, αρχικά τροποποιούμε την open ώστε να πυροδοτείται η αντίστοιχη open της native κρυπτογραφικής συσκευής. Μέσω του προτύπου επικοινωνίας, στέλνουμε την εντολή VIRTIO_CRYPTO_SYSCALL_OPEN και περιμένουμε το αποτέλεσμα από το backend. Το αποτέλεσμα είναι ο file descriptor της native συσκευής τον οποίο αποθηκεύουμε για future reference στο πεδίο private_data του ανοικτού αρχείου. Στο ανοικτό αρχείο έχουμε πρόσβαση μέσω της get_crypto_chrdev_by_minor().

Αντίστοιχα στο close() μέρος του τροποποιημένου driver (λειτουργία VIRTIO_CRYPTO_SYSCALL_CLOSE)στέλνουμε τον fd που επιθυμούμε να κλείσουμε και απελευθερώνουμε όποια δομή δεν απαιτείται πλέον.

Για τη κύρια λειτουργικότητα του frontend, τροποποιούμε την συνάρτηση ioctl με αφετηρία τον δοσμένο κώδικα ώστε σε κάθε ένα απο τα operations που εκτελούμε να μεταφέρονται τα κατάλληλα μηνύματα. Σε κάθε περίπτωση, στέλνουμε την εντολή VIRTIO_CRYPTO_SYSCALL_IOCTL μέσω της syscall_type scatter gather list.

Στη περίπτωση έναρξης συνόδου με τη κρυπτογραφική συσκευή, λαμβάνουμε απο το userspace του guest το αντίστοιχο session (κάθε τέτοια δοσοληψία απαιτεί τη χρήση της copy_from_user()) και όποια άλλη πληροφορία αναλύσαμε παραπάνω για την έναρξη συνόδου με cryptodev API και τα αποστέλλουμε στο backend μέρος.

Στη περίπτωση λήξης συνόδου λαμβάνουμε τη δομή sess απο το userspace του guest και την αποστέλουμε για finalize στο backend μερος.

Στη περίπτωση κρυπτογράφησης/αποκρυπτογράφησης λαμβάνουμε απο τον χρήστη τη δομή cryp καθώς και κάθε επιμέρους πεδίο της (iv,key και όσα αναφέραμε προηγουμένως), και τα αποστέλλουμε για την εκτέλεση της αντίστοιχης λειτουργίας στο backend. Έπειτα αναμένουμε μέχρι να επιστραφούν τα αποτελέσματα της κάθε διαδικασίας και τα επιστρέφουμε με τη κατάλληλη copy_to_user() στον χρήστη.

Τέλος, πρέπει να αναφέρουμε πως σε κάθε περίπτωση επιθυμούμε να μην αλλάζει το interface του νέου μας οδηγού, οπότε και σε κάθε operation επιστρέφεται με copy_to_user() το αποτέλεσμα (host_ret) ώστε να εξακολουθεί να λειτουργέι το error checking των userspace προγραμμάτων που έχουμε απο τα προηγούμενα ζητούμενα.

Περιγραφή Backend

Σε αυτό το τμήμα του οδηγού, οφείλουμε να ερμηνεύουμε τις κατάλληλες εντολές από τα virt queues και, αφού χρησιμοποιήσουμε τη native /dev/crypto συσκευή να επιστρέψουμε τα αποτελέσματα πάλι μέσω των scatter gather lists στο qemu VM. Για κάθε ένα απο τα VIRTIO_* macros επιτελούνται τα εξής:

Κατά το open() ανοίγουμε όπως και στο προηγούμενο ζητούμενο την συσκευή και επιστρέφουμε τον κατάλληλο file descriptor ώστε το εικονικό μηχάνημα να γνωρίζει σε ποια συσκευή θα καλέσει αργότερα εντολές και το close(). Αντίστοιχα κατά τη close() κλείνουμε τη συσκευή και επιστρέφουμε το αποτέλεσμα για το error checking αν όλα πήγαν ως πρέπει.

Κατά την ioctl() ελέγχουμε το command που επιθυμεί να εκτελέσει ο χρήστης και σε κάθε περίπτωση λαμβάνουμε το cryp struct προς επεξεργασία, εκτελούμε όπως το προηγούμενο ζητούμενο και στη συνέχεια επιστρέφουμε το cryp στο VM μαζί με το αντίστοιχο return value για το error checking.

Ο κώδικας του συγκεκριμένου ερωτήματος παρατίθεται στη συνέχεια:

Frontend

```
/*
 * crypto-chrdev.c
 *
 * Implementation of character devices
 * for virtio-crypto device
 *
 * Vangelis Koukis <vkoukis@cslab.ece.ntua.gr>
 * Dimitris Siakavaras <jimsiak@cslab.ece.ntua.gr>
```

```
 * Stefanos Gerangelos <sgerag@cslab.ece.ntua.gr>
 *
 */
#include <linux/cdev.h>
#include <linux/module.h>
#include <linux/poll.h>
#include <linux/sched.h>
#include <linux/semaphore.h>
#include <linux/virtio.h>
#include <linux/virtio_config.h>
#include <linux/wait.h>

#include "crypto-chrdev.h"
#include "crypto.h"
#include "debug.h"

#include "cryptodev.h"

#define MSG_LEN 100 //may never get used

/*
 * Global data
 */
struct cdev crypto_chrdev_cdev;

/**
 * Given the minor number of the inode return the crypto device
 * that owns that number.
 **/
static struct crypto_device *get_crypto_dev_by_minor(unsigned int minor) {
    struct crypto_device *crdev;
    unsigned long flags;

    debug("Entering");

    spin_lock_irqsave(&crdrvdata.lock, flags);
    debug("LOC");
    list_for_each_entry(crdev, &crdrvdata.devs, list) {
        debug(" | T ");
        if (crdev->minor == minor)
        {debug("Found");
                goto out;
        }
    }
    crdev = NULL;

out:
    spin_unlock_irqrestore(&crdrvdata.lock, flags);

    debug("Leaving");
```

```
    return crdev;
}

/************************************
 * Implementation of file operations
 * for the Crypto character device
 ************************************/

static int crypto_chrdev_open(struct inode *inode, struct file *filp) {

    int ret = 0;
    int err;
    //Better have fixed sgs indexes using macros than hardcoded counters
    unsigned int len, input_num=0, output_num=0;
    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");

    ret = -ENODEV;
    if ((ret = nonseekable_open(inode, filp)) < 0)
        goto fail;

    /* Associate this open file with the relevant crypto device. */
    debug("device with %u minor", iminor(inode));
    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;
    }

//    sema_init(&crdev->lock,1);
    crof = kmalloc(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 = kmalloc(sizeof(*syscall_type), GFP_KERNEL);
    *syscall_type = VIRTIO_CRYPTO_SYSCALL_OPEN;

    host_fd = kmalloc(sizeof(*host_fd), GFP_KERNEL);
    *host_fd = -1;


    sg_init_one(&syscall_type_sg, syscall_type, sizeof(*syscall_type));
    sgs[output_num] = &syscall_type_sg;

    output_num++;
    sg_init_one(&host_fd_sg, host_fd, sizeof(*host_fd));
    sgs[output_num + input_num] = &host_fd_sg;
    input_num++;
    /* Going into critical section. We need to lock */

    if (down_interruptible(&crdev->lock))
        return -ERESTARTSYS;

    err = virtqueue_add_sgs(crdev->vq, sgs, output_num, input_num, &syscall_type_sg,
                GFP_ATOMIC);
    virtqueue_kick(crdev->vq);
    while (virtqueue_get_buf(crdev->vq, &len) == NULL)
        /* do nothing */;

    up(&crdev->lock);

    /*
     ** Unlock the Semaphore.
     ** If host failed to open() return -ENODEV.
     */

    if (*host_fd < 0) {
        ret = -ENODEV;
        goto fail;
    }

    crof->host_fd = *host_fd;

    kfree(syscall_type);
    kfree(host_fd);

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

static int crypto_chrdev_release(struct inode *inode, struct file *filp) {
    int ret = 0;
```

```
int err;
unsigned int len, output_num=0;
struct crypto_open_file *crof = filp->private_data;
struct crypto_device *crdev = crof->crdev;
unsigned int *syscall_type;
int *host_fd;
struct scatterlist syscall_type_sg, host_fd_sg, *sgs[2];
debug("Entering");

/**
 * Send data 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;


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

sg_init_one(&host_fd_sg, host_fd, sizeof(*host_fd));
sgs[output_num] = &host_fd_sg;
output_num++;
/**
 * Wait for the host to process our data.
 **/

/*
 ** Going into critical section.
 */

if (down_interruptible(&crdev->lock))
    return -ERESTARTSYS;

err = virtqueue_add_sgs(crdev->vq, sgs, output_num, 0, &syscall_type_sg,
           GFP_ATOMIC);
virtqueue_kick(crdev->vq);
while (virtqueue_get_buf(crdev->vq, &len) == NULL)
    /* do nothing */;

up(&crdev->lock);

/* Unlock and return */

if (*host_fd < 0)
    ret = -1;
```

```
    kfree(syscall_type);
    kfree(host_fd);
    kfree(crof);
    debug("Leaving");
    return ret;
}

static long crypto_chrdev_ioctl(struct file *filp, unsigned int cmd,
        unsigned long arg) {
    long ret = 0;
    unsigned long flags;
    int err, *host_fd, *host_ret;
    unsigned int output_num=0, input_num=0, len, *cmdpointer, *syscall_type;
    unsigned char *source, *dest, *iv, *session_key, *temp, *ses_temp;
    uint32_t *ses_id;

    struct crypto_open_file *crof = filp->private_data;
    struct crypto_device *crdev = crof->crdev;
    struct virtqueue *vq = crdev->vq;
    struct scatterlist syscall_type_sg, output_msg_sg, input_msg_sg, cmd_sg,
                       session_sg, host_fd_sg, host_ret_sg, session_id_sg, cryp_src_sg,
                       cryp_dst_sg, cryp_iv_sg, cryp_op_sg, sessionkey_sg, *sgs[13];
    struct session_op *session;
    struct crypt_op *cryp;

    debug("Entering");

    session_key = NULL;
    source = NULL;
    dest = NULL;
    iv = NULL;
    temp = NULL;
    ses_temp = NULL;
    /* Do memory allocations */

    ses_id = kmalloc(sizeof(*ses_id), GFP_KERNEL);

    cmdpointer = kmalloc(sizeof(*cmdpointer), GFP_KERNEL);
    *cmdpointer = cmd;

    host_ret = kmalloc(sizeof(*host_ret), GFP_KERNEL);

    syscall_type = kmalloc(sizeof(*syscall_type), GFP_KERNEL);
    *syscall_type = VIRTIO_CRYPTO_SYSCALL_IOCTL;

    session = kmalloc(sizeof(*session), GFP_KERNEL);

    cryp = kmalloc(sizeof(*cryp), GFP_KERNEL);

    host_fd = kmalloc(sizeof(*host_fd), GFP_KERNEL);
```

```
    *host_fd = crof->host_fd;

    /**
     *  These are common to all ioctl commands.
     **/

    sg_init_one(&syscall_type_sg, syscall_type, sizeof(*syscall_type));
    sgs[output_num] = &syscall_type_sg;

    output_num++;

    sg_init_one(&host_fd_sg, host_fd, sizeof(*host_fd));
    sgs[output_num] = &host_fd_sg;

    output_num++;

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

                sg_init_one(&cmd_sg, cmdpointer, sizeof(*cmdpointer));
                sgs[output_num] = &cmd_sg;
                output_num++;
                err = copy_from_user(session, (struct session_op *)arg, sizeof(*session));

                if (err) {
                        debug("CAN'T GET SESSION FROM IOCTL");
                        ret = -1;
                        goto fail;
                }

                session_key = kmalloc(session->keylen * sizeof(char), GFP_KERNEL);
                if (!session_key) {
                        ret = -ENOMEM;
                        goto fail;
                }

                if (copy_from_user(session_key, session->key, sizeof(char) *
session->keylen)) {
                        debug("CAN'T GET KEY");
                        ret = -1;
                        goto fail;
                }
                ses_temp=session_key;
                sg_init_one(&sessionkey_sg, session_key, sizeof(char) * session->keylen);
                sgs[output_num] = &sessionkey_sg;
                output_num++;
```

```
        sg_init_one(&session_sg, session, sizeof(*session));
        sgs[output_num + input_num] = &session_sg;
        input_num++;

        sg_init_one(&host_ret_sg, host_ret, sizeof(*host_ret));
        sgs[output_num + input_num] = &host_ret_sg;
        input_num++;

        break;

case CIOCFSESSION: // Close Session
        debug("CIOCFSESSION");

        sg_init_one(&cmd_sg, cmdpointer, sizeof(*cmdpointer));
        sgs[output_num] = &cmd_sg;
        output_num++;

        if (copy_from_user(ses_id, (uint32_t *)arg, sizeof(*ses_id))) {
                debug("CAN'T GET SESSION ID");
                ret = -1;
                goto fail;
        }

        sg_init_one(&session_id_sg, ses_id, sizeof(*ses_id));
        sgs[output_num] = &session_id_sg;
        output_num++;

        sg_init_one(&host_ret_sg, host_ret, sizeof(*host_ret));
        sgs[output_num + input_num] = &host_ret_sg;
        input_num++;
        break;

case CIOCCRYPT: // Encrypt/Decrypt
        debug("CIOCCRYPT");

        sg_init_one(&cmd_sg, cmdpointer, sizeof(*cmdpointer));
        sgs[output_num] = &cmd_sg;
        output_num++;

        if (copy_from_user(cryp, (struct crypt_op *)arg, sizeof(*cryp))) {
                debug("CANT GET CRYPT OPER check fields");
                ret = -1;
                goto fail;
        }

        sg_init_one(&cryp_op_sg, cryp, sizeof(*cryp));
        sgs[output_num] = &cryp_op_sg;
        output_num++;
```

```
            source = kmalloc(cryp->len * sizeof(char), GFP_KERNEL);
            if (copy_from_user(source, cryp->src, cryp->len * sizeof(char))) {
                    debug("CANT GET SOURCE DATA");
                    ret = -1;
                    goto fail;
            }

            sg_init_one(&cryp_src_sg, source, cryp->len * sizeof(char));
            sgs[output_num] = &cryp_src_sg;
            output_num++;

            iv = kmalloc(16 * sizeof(char), GFP_KERNEL);
            if (copy_from_user(iv, cryp->iv, 16 * sizeof(char))) {
                    debug("CANT GET IV");
                    ret = -1;
                    goto fail;
            }

            sg_init_one(&cryp_iv_sg, iv, 16 * sizeof(char));
            sgs[output_num] = &cryp_iv_sg;
            output_num++;

            temp = cryp->dst;
            dest = kmalloc(cryp->len * sizeof(char), GFP_KERNEL);
            sg_init_one(&cryp_dst_sg, dest, cryp->len * sizeof(char));
            sgs[output_num + input_num] = &cryp_dst_sg;
            input_num++;

            sg_init_one(&host_ret_sg, host_ret, sizeof(*host_ret));
            sgs[output_num + input_num] = &host_ret_sg;
            input_num++;
            break;

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

/**
 * Wait for the host to process our data.
 **/

/* Lock!!! Critical section */

if (down_interruptible(&crdev->lock))
    return -ERESTARTSYS;

err = virtqueue_add_sgs(vq, sgs, output_num, input_num, &syscall_type_sg,
            GFP_ATOMIC);
virtqueue_kick(vq);
```

```
    while (virtqueue_get_buf(vq, &len) == NULL)
        /* do nothing */;

    up(&crdev->lock);

    // Return to user the thing he needs!

    switch (cmd) {
        case CIOCGSESSION: // Get Session
                debug("Return from CIOCGSESSION");
                session->key = ses_temp;
                if (copy_to_user((struct session_op *)arg, session, sizeof(*session))) {
                        debug("FAILED TO START A SESSION");
                        ret = -1;
                        goto fail;
                }

                kfree(session_key);
                break;

        case CIOCFSESSION: // Close Session

                debug("Return from CIOCFSESSION");
                break;

        case CIOCCRYPT: // Encrypt/Decrypt
                debug("Return from CIOCCRYPT");

                if (copy_to_user(temp, dest, cryp->len * sizeof(char))) {
                        debug("FAILED TO ENCRYPT/DECRYPT YOUR DATA");
                        ret = -1;
                        goto fail;
                }

                kfree(source);
                kfree(iv);
                kfree(dest);
                break;
    }

    kfree(ses_id);
    kfree(cmdpointer);
    kfree(host_fd);
    kfree(host_ret);
    kfree(session);
    kfree(cryp);
    kfree(syscall_type);
fail:

    debug("Leaving");
```

```
    return ret;
}

static ssize_t crypto_chrdev_read(struct file *filp, char __user *usrbuf,
        size_t cnt, loff_t *f_pos) {
    debug("Entering");
    debug("Leaving");
    return -EINVAL;
}

static struct file_operations crypto_chrdev_fops = {
    .owner = THIS_MODULE,
    .open = crypto_chrdev_open,
    .release = crypto_chrdev_release,
    .read = crypto_chrdev_read,
    .unlocked_ioctl = crypto_chrdev_ioctl,
};

int crypto_chrdev_init(void) {
    int ret;
    dev_t dev_no;
    unsigned int crypto_minor_cnt = CRYPTO_NR_DEVICES;

    debug("Initializing character device...");
    cdev_init(&crypto_chrdev_cdev, &crypto_chrdev_fops);
    crypto_chrdev_cdev.owner = THIS_MODULE;

    dev_no = MKDEV(CRYPTO_CHRDEV_MAJOR, 0);
    ret = register_chrdev_region(dev_no, crypto_minor_cnt, "crypto_devs");
    if (ret < 0) {
        debug("failed to register region, ret = %d", ret);
        goto out;
    }
    ret = cdev_add(&crypto_chrdev_cdev, dev_no, crypto_minor_cnt);
    if (ret < 0) {
        debug("failed to add character device");
        goto out_with_chrdev_region;
    }

    debug("Completed successfully");
    return 0;

out_with_chrdev_region:
    unregister_chrdev_region(dev_no, crypto_minor_cnt);
out:
    return ret;
}

void crypto_chrdev_destroy(void) {
```

```
    dev_t dev_no;
    unsigned int crypto_minor_cnt = CRYPTO_NR_DEVICES;

    debug("entering");
    dev_no = MKDEV(CRYPTO_CHRDEV_MAJOR, 0);
    cdev_del(&crypto_chrdev_cdev);
    unregister_chrdev_region(dev_no, crypto_minor_cnt);
    debug("leaving");
}
```

Backend

```
/*
 * Virtio Crypto Device
 *
 * Implementation of virtio-crypto qemu backend device.
 *
 * Dimitris Siakavaras <jimsiak@cslab.ece.ntua.gr>
 * Stefanos Gerangelos <sgerag@cslab.ece.ntua.gr>
 *
 */

#include "hw/virtio/virtio-crypto.h"
#include "hw/virtio/virtio-serial.h"
#include <crypto/cryptodev.h>
#include <fcntl.h>
#include <qemu/iov.h>
#include <sys/ioctl.h>
#include <sys/stat.h>
#include <sys/types.h>

static uint32_t get_features(VirtIODevice *vdev, uint32_t features) {
        DEBUG_IN();
        return features;
}

static void get_config(VirtIODevice *vdev, uint8_t *config_data) { DEBUG_IN(); }

static void set_config(VirtIODevice *vdev, const uint8_t *config_data) {
        DEBUG_IN();
}

static void set_status(VirtIODevice *vdev, uint8_t status) { DEBUG_IN(); }

static void vser_reset(VirtIODevice *vdev) { DEBUG_IN(); }
```

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

        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;
        int *host_fd;
        switch (*syscall_type) {
        case VIRTIO_CRYPTO_SYSCALL_TYPE_OPEN:
        DEBUG("VIRTIO_CRYPTO_SYSCALL_TYPE_OPEN");
        host_fd = elem.in_sg[0].iov_base; // host_fd is now a pointer to host_fd
                                          // , W flag so we will return it
        *host_fd = open("/dev/crypto", O_RDWR);
        if ((*host_fd) < 0) {
                perror("open(/dev/crypto)");
                return;
        }
        break;

        case VIRTIO_CRYPTO_SYSCALL_TYPE_CLOSE:
        DEBUG("VIRTIO_CRYPTO_SYSCALL_TYPE_CLOSE");
        host_fd = elem.out_sg[1].iov_base; // host_fd is now a pointer to
                                          // host_fd ,we have to read it
        if (close(*host_fd) < 0) {
                *host_fd = -1;
                perror("close");
                return;
        }
        break;

        case VIRTIO_CRYPTO_SYSCALL_TYPE_IOCTL:
        DEBUG("VIRTIO_CRYPTO_SYSCALL_TYPE_IOCTL");
        /* ?? */
        // unsigned char *output_msg = elem.out_sg[1].iov_base;
        // unsigned char *input_msg = elem.in_sg[0].iov_base;
        // memcpy(input_msg, "Host: Welcome to the virtio World!", 35);
        // printf("Guest says: %s\n", output_msg);
        // printf("We say: %s\n", input_msg);

        unsigned int *ioctl_cmd = elem.out_sg[2].iov_base;              // common for all
three
        int *host_return = elem.in_sg[1].iov_base; // -1 if something goes wrong
```

```
switch (*ioctl_cmd) {
case CIOCGSESSION:
        DEBUG("Backend starting a crypto session");
        host_fd = elem.out_sg[1].iov_base;
        unsigned char *session_key = elem.out_sg[3].iov_base;
        struct session_op *session_op_st = elem.in_sg[0].iov_base;
        session_op_st->key = session_key;
        if (ioctl(*host_fd, CIOCGSESSION, session_op_st)) {
        *host_return = -1;
        perror("Backend crypto session fail starting");

        } else {
        *host_return = 0;
        DEBUG("Backend crypto session starting succesfully");
        }
        break;

case CIOCFSESSION:
        DEBUG("Backend ending a crypto session");
        host_fd = elem.out_sg[1].iov_base;
        uint32_t *ses_id = elem.out_sg[3].iov_base;
        if (ioctl(*host_fd, CIOCFSESSION, ses_id)) {
        *host_return = -1;
        perror("Backend crypto session fail ending");
        } else {
        *host_return = 0;
        DEBUG("Backend crypto session ending succesfully");
        }
        break;
case CIOCCRYPT:
        DEBUG("Backend encrypting/decrypting");
        host_fd = elem.out_sg[1].iov_base;
        struct crypt_op *crypt_operands = elem.out_sg[3].iov_base;
        unsigned char *source = elem.out_sg[4].iov_base;
        unsigned char *iv = elem.out_sg[5].iov_base;
        unsigned char *destination = elem.in_sg[0].iov_base;
        crypt_operands->src = source;
        crypt_operands->dst = destination;
        crypt_operands->iv = iv;

        if (ioctl(*host_fd, CIOCCRYPT, crypt_operands)) {
        perror("Backend fail ioctl(CIOCCRYPT)");
        *host_return = -1;
        } else {
        *host_return = 0;
        DEBUG("Backend encrypting/decrypting succesful");
        }
        break;
```

```
        default:
                DEBUG("Unknown ioctl command");
                break;
        }
        break;

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

        virtqueue_push(vq, &elem, 0);
        virtio_notify(vdev, vq);
}

static void virtio_crypto_realize(DeviceState *dev, Error **errp) {
        VirtIODevice *vdev = VIRTIO_DEVICE(dev);

        DEBUG_IN();

        virtio_init(vdev, "virtio-crypto", 13, 0);
        virtio_add_queue(vdev, 128, vq_handle_output);
}

static void virtio_crypto_unrealize(DeviceState *dev, Error **errp) {
        DEBUG_IN();
}

static Property virtio_crypto_properties[] = {
        DEFINE_PROP_END_OF_LIST(),
};

static void virtio_crypto_class_init(ObjectClass *klass, void *data) {
        DeviceClass *dc = DEVICE_CLASS(klass);
        VirtioDeviceClass *k = VIRTIO_DEVICE_CLASS(klass);

        DEBUG_IN();
        dc->props = virtio_crypto_properties;
        set_bit(DEVICE_CATEGORY_INPUT, dc->categories);

        k->realize = virtio_crypto_realize;
        k->unrealize = virtio_crypto_unrealize;
        k->get_features = get_features;
        k->get_config = get_config;
        k->set_config = set_config;
        k->set_status = set_status;
        k->reset = vser_reset;
}

static const TypeInfo virtio_crypto_info = {
```

```
        .name = TYPE_VIRTIO_CRYPTO,
        .parent = TYPE_VIRTIO_DEVICE,
        .instance_size = sizeof(VirtCrypto),
        .class_init = virtio_crypto_class_init,
};

static void virtio_crypto_register_types(void) {
        type_register_static(&virtio_crypto_info);
}

type_init(virtio_crypto_register_types)
```