Σιγλίδης Γιάννης
Κωνσταντινίδης Ορέστης
Ομάδα Α22

Εργαστήριο Λειτουργικών Συστημάτων :

Κρυπτογραφική συσκευή VirtIO
για QEMU-KVM

3η Άσκηση για το μάθημα Εργαστήριο Λειτουργικών Συστημάτων.
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Διδάσκοντες : Ν. Κοζύρης, Π. Τσανάκας

1 Εισαγωγή

Η εργασία αυτή αποτελούνταν από τρία ζητούμενα. Στο πρώτο καλούμασταν να δημιουργήσουμε ένα εργαλείο *chat* το οποίο θα λειτουργεί πάνω από πρωτόκολλο TCP/IP με χρήση των εργαλείων που παρέχουν τα BSD *sockets*.
Το δεύτερο ζητούμενο αφορούσε την υλοποίηση κρυπτογράφησης πάνω από την υλοποιηση του πρώτου ζητούμενου, με χρήση της συσκευής cryptodev όπως προκύπτει από τον *driver* του *cryptodev-linux*.
Το τρίτο και τελευταίο ζητούμενο ήταν η υλοποίηση του σχήματος παραεικονικοποίησης μέσω του virtIO ώστε να γίνει εφικτή η εκτέλεση της εφαρμογής *chat* του πρώτου ζητούμενου με κρυπτογράφηση μέσα σε *vm*, αναθέτοντας την κρυπτογράφηση εκτός *vm*, στον *host*.

2 Z1: Εργαλείο chat πάνω από TCP/IP *sockets*

Το ζητούμενο της άσκησης Z1 υλοποιήθηκε επάνω στον ήδη υπάρχοντα κωδικα του *socket-client*, *socket-server* όπως αυτοί παραδόθηκαν ως βοηθητικά αρχεία. Το αρχικό μοντέλο που ακολουθήθηκε αφορούσε την υλοποίηση ενός *chat* μεταξύ ενός *server* και ενός *client*. Κάτι τέτοιο έγινε περισσότερο για την εξοικείωση με τον υπάρχοντα κώδικα και όχι ως μία τελική υλοποίηση. Μετά από παρότρυνση του οδηγού αποφασίσαμε να υλοποιήσουμε ένα μοντέλο IRC κατά το οποίο πολλοί *clients* συνδέονται στον ίδιο *server*, ενώ ο *server* λειτουργεί σε ρόλο "διακομιστή", παρακολουθώντας την αποστολή δεδομένων του κάθε χρήστη, ενώ εξασφαλίζει ότι αυτά θα γίνουν *broadcast* σε όλους τους υπόλοιπους.

2.1 Socket-client

Το εκτελέσιμο socket-client δέχεται ως όρισμα την ip του server στον οποίο θέλει ο χρήστης να συνδεθεί, καθώς και την port στην οποία ο server "ακούει", προκειμένου να εκτελέσει επιτυχώς μία IRC συνομιλία.
Για να ξεκινήσει η συνομιλία ο client ανοίγει στην γραμμή 86 ένα socket τύπου PF_INET (ipv4) με πρωτόκολλο επικοινωνίας που καθορίζεται από την παράμετρο SOCK_STREAM (TCP/IP). Στην συνέχεια στην γραμμή 93 αναζητά τον server με το hostname του στο DNS, χρησιμοποιώντας τη συνάρτηση *gethostbyname()*.
Για να ολοκληρωθεί η συνδεση απαιτείται η επιτυχία της εντολής connect που φαίνεται στη γραμμή 103, μετά από την οριστικοποίηση των παραμέτρων της μεταβλητής *sa* που είναι τύπου *struct sockaddr_in*.
Προκειμένου να εξασφαλίσουμε την ταυτόχρονη λήψη και αποστολή μηνυμάτων στον *server*, χρησιμοποιήσαμε την συνάρτηση *select* της οποίας η κλήση φαίνεται στη γραμμή 114. Η συνάρτηση *select* στο σημείο που καλείται, διακόπτει την εκτέλεση του προγράμματος μέχρι τουλάχιστον ένας από τους *client/server* να έχει "γράψει" στον περιγραφητή αρχείου του. Για τον *client* αυτός είναι ο **0** καθώς αντιστοιχεί στο stdin ενώ για τον για τον server είναι ο sd όπως ορίζεται κατά την εκτέλεση της εντολής socket στη γραμμή 86.
Με την χρήση της συνάρτηση FD_ISSET αποφαινόμαστε ποιος/ποιοι εκ των δύο περιγραφητών αρχείων ενεργοποίηθηκε και με κατάλληλη χρήση των συναρτήσεων *insist_write()* και *insist_read()* τυπώνουμε το μήνυμα που στέλνει ο *server* στον *client* ή στέλνουμε αυτό που γράφει ο *client* στο *stdin*, στον *server*. Η όλη επικοινωνία βρίσκεται σε ένα *infinite loop* το οποίο τερματίζει ή κάνοντας *ctrl-c* είτε στέλνοντας EOF (*ctrl-d*) από τη μεριά του χρήστη είτε αν ο *server* κλείσει το δίαυλο επικοινωνίας από τη μεριά του.

2.2 *Socket-server*

Από την μεριά του *server*, γίνεται μια αντίστοιχη διαδικασία, όπου η βασική διαφορά για το " *setup*" είναι ότι ο *server* έχει μια στατική *ip* και μια *standard port* ώστε να μπορούν οι *clients* να ξέρουν που να συνδεθούνε.
Έτσι αφού ορίσει *socket* στη γραμμή 165, σε έναν περιγραφητή *sd*, αντιστοιχίζει μέσω της συνάρτησης *bind* την στατική *ip* και το port που "ακούει" στον περιγραφητή αυτόν, όπως φαίνεται στην γραμμή 176.

Κάνοντας κάτι τέτοιο, μέσω της συνάρτησης *listen*, στην γραμμή 183, "ακούει" για *incoming connections*, οι οποίες μέσω της σταθεράς TCP_BACKLOG μπορούν να αναμένουν σε ουρά, μεγέθους το πολύ 5. Όσον αφορά την *select*, ο *server* καλείται αυτή την φορά να επιλέγει μεταξύ όλων των *clients* που είναι συνδεδεμένοι σε αυτόν καθώς και της διεύθυνσης στην οποία "ακούει", όπως έχει αποτυπωθεί στον περιγραφητή *sd*. Όποτε ενεργοποιείται η τελευταία, ο *server*, μέσω της συνάρτησης *accept()* στην γραμμή 207, αποδέχεται μία νέα σύνδεση για την οποία αποθηκεύει σε έναν πίνακα τον περιγραφητή της, το όνομα του χρήστη όπως δόθηκε κατά την κλήση της συνάρτησης *getname()* και τέλος, προσθέτει τον νέο αυτόν περιγραφητή στο υπάρχον σύνολο από το οποίο θα γίνεται *select()*, προκειμένου να παρατηρεί εισερχόμενα μηνύματα. Στην περίπτωση που ο *server* λάβει ένα εισερχόμενο μήνυμα, το στέλνει σε όλους τους υπόλοιπους χρήστες που είναι ήδη συνδεδεμένοι (παρατηρώντας το από την εντολή FD_ISSET, στην γραμμή 265), ενώ στην περίπτωση που ο *client* κλείσει τον δίαυλο επικοινωνίας, τον αφαιρεί από το σύνολο των συνδεδεμένων χρηστών και "καθαρίζει" την αντίστοιχη θέση του πίνακα. Για να παρατηρεί όλους τους χρήστες καθώς και τον ίδιο τον server, αποθηκεύει σε μια μεταβλητή *fd_max*, τον μέγιστο *file descriptor* που έχει παρατηρηθεί, ενώ η όλη διαδικασία της αμφίδρομης επικοινωνίας μεταξύ *server/client* βρίσκεται σε ένα *infinite loop* κάνοντας απαραίτητο για τον τερματισμό του εκτελεσίμου *socket-server* το *ctrl-c*.

3 Z2: Κρυπτογραφημένο chat πάνω από TCP/IP

Το κρυπτογραφημένο *chat* έγινε πάνω στο παραπάνω μοντέλο που υλοποιήθηκε στο Z1. Από την μεριά του *client*, δημιουργούμε ένα κλειδί με τον *server* με τρόπο που ορίζεται από το πρωτόκολλο του *crypto-dev* η υλοποίηση του οποίου φαίνεται στο βοηθητικό αρχείο, *crypto-test.c*. Με το κλειδί αυτό κρυπτογραφούμε τα δεδομένα που στέλνουμε στον *server*, ενώ συμμετρικά αποκρυπτογραφούμε το μήνυμα του *server*, το οποίο γίνεται μέσω εντολών *ioctl()* επάνω σε έναν περιγραφητή αρχείου *crypto_fd* στην συσκευή */dev/crypto*. Από την μεριά του *server*, κάθε φορά που ένας *client* συνδέεται, δημιουργείται ένα μοναδικό κλειδί, που αποθηκεύεται σε ένα νέο πεδίο της δομής *User* στον πίνακα *peer*. Έτσι, κάθε φορά που ο *server* κάνει *broadcast* ένα μήνυμα ενός *client*, σε όλους τους υπόλοιπους, το κρυπτογραφεί με το αντίστοιχο κλειδί του καθενός.

4 Z3: Υλοποίηση συσκευής *cryptodev* με *VirtIO*

Σκοπός του Z3, είναι η υλοποίηση ενός σχήματος παραεικονικοποίησης μέσω του πρωτοκόλλου *virtIO* με τον σχεδιασμό ενός *backend userspace* προγράμματος, τμήμα του εκτελεσίμου της συσκευής εικονικοποίησης *qemu* και ενός *frontend guest kernel space driver*. Σκοπός του *frontend driver* είναι η δημιουργία της ψευδαίσθησης στον χρήστη του *vm*, ύπαρξης συσκευών κρυπτογράφησης τύπου *cryptodev*. Μέσω του πρωτοκόλλου *virtIO*, το *frontend* μέρος επικοινωνεί με το *backend* διαβιβάζοντάς του με κατάλληλο τρόπο τα δεδομένα που προκύπτουν από τις κλήσεις συστήματος *open, close* και *ioctl* που κάνει ο χρήστης σε *guest userspace*. Το τμήμα του *backend* παραλαμβάνει τα δεδομένα από το *frontend*, όπως μεταβιβάζονται μέσω του *virtIO* πρωτοκόλλου και πραγματοποιεί τις αντίστοιχες κλήσεις συστήματος ως *userspace host*. Το αποτέλεσμα αυτής της πράξης, επιστρέφεται από το *backend* στο *frontend* τμήμα, ξανά μέσω του πρωτοκόλλου *virtIO*, και ολοκληρώνεται η διαδικασία της κρυπτογράφησης χωρίς ο *guest* χρήστης να "καταλάβει" ότι τα δεδομένα αλλάξανε περιβάλλον. Η αίσθηση του *guest* χρήστη είναι, ότι στο εικονικό περιβάλλον υπάρχει μια συσκευή που κρυπτογραφεί τα δεδομένα του.

4.1 *Frontend*

Από την μεριά του *frontend*, χρησιμοποιήσαμε τον ήδη υπάρχοντα σκελετό, όπως δόθηκε στον βοηθητικό κώδικα, και συγκεκριμένα τροποποιήσαμε κατάλληλα το αρχείο *crypto-chrdev.c*. Οι προσθήκες που χρειάστηκε να κάνουμε, αφορούσαν τις συναρτήσεις *crypto_chrdev_open*, *crypto_chrdev_release* και

crypto_chrdev_ioctl. Οι προσθήκες αφορούσαν το κομμάτι της μεταβίβασης δεδομένων όπου χρησιμοποιήθηκαν οι *scattergather lists* μέσω του πρωτοκόλλου των *virtqueues* του *virtIO*, προκειμένου να μεταφερθεί η πληροφορία όπως φαίνεται στην σελίδα 14 του οδηγού.

4.2 *Backend*

Από την μεριά του *backend* τμήματος, που βρίσκεται στον κώδικα του *qemu*, με βάση την δεδομένη μορφοποίηση της εισόδου όπως αυτή διαβιβάζεται μέσω των δομών των *virtqueues* από το *vm*, πραγματοποιήσαμε τις κατάλληλες κλήσεις συστήματος και τροποποιήσαμε τα πεδία των *scatter-gather lists* που είχαν δικαιώματα εγγραφής. Κατά αυτόν τον τρόπο, το *backend* τμήμα είχε το ρόλου του "σκλάβου", ο οποίος εκτελούσε τις διαταγές, όπως "προστάζονταν" από το *frontend* τμήμα. Ως συνέπεια αυτού, το *frontend* ήταν το υπεύθυνο για το "τι πρέπει να γίνει" και το *backend* τμήμα ήταν υπεύθυνο για το "πως πρέπει να γίνει" η κατάλληλη διαδικασία ώστε να εκτελεστεί σωστά στο " *overview*" η κρυπτογράφηση. Κατα συνέπεια το *frontend* καλούνταν να θυμάται τον περιγραφητή αρχείου που έχει ανοίξει το *backend* τμήμα καθώς και να παρέχει προστασία μέσω *spinlock* για παράλληλη εγγραφή στο ίδιο αρχείο από μία διεργασία και κάποιο της παιδί.

Επειδή οι προσθήκες στα παραπάνω κομμάτια ήταν αρκετά εκτεταμένες δεν γίνεται σε αυτήν την αναφορά να τις περιγράψουμε όλες και με γνώμονα ότι ελέγχθηκαν από τον βοηθό του εργαστηρίου κατά την εξέταση, θεωρήσαμε ότι δεν είναι απαραίτητο να αναλυθούν λεπτομερώς. Παρ' όλα αυτά ο κώδικας θα ενσωματωθεί στην αναφορά και ο αναγνώστης μπορεί να ελέγξει τις λεπτομέρειες ο ίδιος.

socket–client–z1.c

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

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

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

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

#include "socket-common.h"

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

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

        return orig_cnt;
```

```
}
ssize_t insist_read(int sd,char buf[MESSAGE_SIZE+230], size_t cnt)
{
        ssize_t ret;
        size_t orig_cnt = cnt;
        char A[MESSAGE_SIZE+230];
        void *c;
        c = A;
        while (cnt > 0) {
                ret = read(sd, c, cnt);
                if (ret < 0)
                        return ret;
                c += ret;
                cnt -= ret;
        }
        strcpy(buf,A);
        return orig_cnt;


}

int main(int argc, char *argv[])
{

        fd_set master;    // master file descriptor list
        fd_set read_fds;  // temp file descriptor list for select()
        int sd, port;
        int i=0;
        int fdmax;
        int count;
        char c;
        ssize_t n;
        char buf[MESSAGE_SIZE+230];
        char *hostname;
        struct hostent *hp;
        struct sockaddr_in sa;

        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");
        FD_SET(sd, &master);
        FD_SET(0, &master);
        fdmax = sd;
        for(;;){
                read_fds = master; // copy it
                fprintf(stdout,"Waiting for a selection...\n");
                if (select(fdmax+1, &read_fds, NULL, NULL, NULL) == -1) {
                    perror("select");
                    exit(4);
```

```
                }
                fprintf(stdout,"A selection has been made!\n");
                if (FD_ISSET(sd, &read_fds)){
                        /*
                        * Let the remote know we're not going to write anything else.
                        * Try removing the shutdown() call and see what happens.
                        */
                        memset(buf,0,sizeof(buf));
                        /* Read answer and write it to standard output */
                        n = read(sd, buf, sizeof(buf));
                        if (n < 0) {
                                perror("read");
                                exit(1);
                        }
                        if (n <= 0) goto out;
                        fprintf(stdout, "Server Replies:\n");
                        if (insist_write(1, buf, n) != n) {
                                perror("write");
                                exit(1);
                        }
                        //fprintf(stdout,"\n");
                }
                if (FD_ISSET(0, &read_fds)){
                        memset(buf,0,sizeof(buf));
                        i=0;
                        fprintf(stdout,"Say Something to Server ::\n");
                        /* Be careful with buffer overruns, ensure NUL-termination */
                        //clearerr(stdin);

                        while((count = fscanf(stdin,"%c",&c))!=-1 && c!='\n' && i<=MESSAGE_SIZE-2){
                                buf[i]=c;
                                i++;
                        }
                        if(count==-1){
                                goto out;
                        }
                        buf[i] = '\0';
                        /* Say something... */
                        if (insist_write(sd, buf, strlen(buf)) != strlen(buf)) {
                                perror("write");
                                exit(1);
                        }
                }
        }
out:
        if (shutdown(sd, SHUT_WR) < 0) {
                        perror("shutdown");
                        exit(1);
        }
        fprintf(stdout, "\nDone.\n");
        return 0;
}
```

socket–server–z1.c

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

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

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

```c

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

#include "socket-common.h"


struct User{
char name[100];
char address[100];
int sd_link;
int init;
};

void * safemalloc(size_t s){
void *cd;
cd = malloc(s);
if(cd == NULL){
        perror("memory");
        exit(1);
}
return cd;
}
/* 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 read_msg(char buf[MESSAGE_SIZE],int newsd){
ssize_t n;
char A[MESSAGE_SIZE];
memset(A,0,sizeof(A));
n = read(newsd, A, MESSAGE_SIZE*sizeof(char));
//printf("READ MSG ::::: A == %s\n",A);
strcpy(buf,A);
return n;
}
int send_msg(int f,int newsd,char A[MESSAGE_SIZE]){
/* if f equals 0 read from stdin */
char ans[MESSAGE_SIZE];
char c;
int i;
if(f == 0){
        printf("Reply ::\n");
        memset(ans,0,sizeof(ans));
        i=0;
        while(fscanf(stdin,"%c",&c)!=0 && c!='\n' && i<=MESSAGE_SIZE-2){
                        ans[i]=c;
                        i++;
        }
        ans[i] = '\0';
}
else{
        strcpy(ans,A);
}
fprintf(stdout, "Your Answer is:\n%s\n",ans);
if (insist_write(newsd,ans,strlen(ans)) != strlen(ans)) {
                perror("write to remote peer failed");
                return -1;
}
return 0;
}

```

```
int getname(struct User *peer){
        ssize_t n;
        int retval;
        struct timeval tv;
        char A[100];

        fd_set master;    // master file descriptor list
        fd_set read_fds;  // temp file descriptor list for select()
        FD_ZERO(&master);    // clear the master and temp sets
        FD_ZERO(&read_fds);
        FD_SET(peer->sd_link, &master);
        tv.tv_sec = WAIT_SEC;
        tv.tv_usec = 0;
        if (insist_write(peer->sd_link, strdup("Who are you?\n"), strlen(strdup("Who are you?\n")))
     == -1) {
                perror("send");
        }
        read_fds = master; // copy it
        fprintf(stdout,"Waiting for a selection... In ... getname()\n");
        retval = select(peer->sd_link+1, &read_fds, NULL, NULL, &tv);
        if(retval == -1){
                perror("select");
                return -1;
        }
        else{
                if(retval){
                        n = read_msg(A,peer->sd_link);
                        if (n <= 0) {
                                if (n < 0){
                                        perror("read from remote peer failed");
                                }
                                else{
                                        fprintf(stderr, "Peer went away\n");
                                }
                                return -1;
                        }
                        sprintf(peer->name,"%s",A);
                        return 1;
                }
                else{
                        printf("Peer with sd : %d on address %s didn't answer get_name question
    within %d seconds.\n",peer->sd_link,peer->address,WAIT_SEC);
                        return 0;
                }
        }
}

int main(void)
{
        fd_set master;    // master file descriptor list
        fd_set read_fds;  // temp file descriptor list for select()
        int fdmax;        // maximum file descriptor number
        char buf[MESSAGE_SIZE];
        struct User * peer;
        char A[MESSAGE_SIZE+200];
        char addrstr[INET_ADDRSTRLEN];
        int sd, newsd;

        peer = (struct User *)safemalloc((PEER_LEN)*sizeof(struct User));

        int i,j;
        ssize_t n;
        socklen_t len;
        struct sockaddr_in sa;

        /* Make sure a broken connection doesn't kill us */
        signal(SIGPIPE, SIG_IGN);
        FD_ZERO(&master);    // clear the master and temp sets
        FD_ZERO(&read_fds);

        /* 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");
```

```
170
171         /* Bind to a well-known port */
172         memset(&sa, 0, sizeof(sa));
173         sa.sin_family = AF_INET;
174         sa.sin_port = htons(TCP_PORT);
175         sa.sin_addr.s_addr = htonl(INADDR_ANY);
176         if (bind(sd, (struct sockaddr *)&sa, sizeof(sa)) < 0) {
177                 perror("bind");
178                 exit(1);
179         }
180         fprintf(stderr, "Bound TCP socket to port %d\n", TCP_PORT);
181
182         /* Listen for incoming connections */
183         if (listen(sd, TCP_BACKLOG) < 0) {
184                 perror("listen");
185                 exit(1);
186         }
187         // add the listener to the master set
188         FD_SET(sd, &master);
189
190         // keep track of the biggest file descriptor
191         fdmax = sd; // so far, it's this one
192         /* Loop forever, accept()ing connections */
193         for (;;) {
194                 read_fds = master; // copy it
195                 fprintf(stdout,"Waiting for a selection...\n");
196                 if (select(fdmax+1, &read_fds, NULL, NULL, NULL) == -1) {
197                     perror("select");
198                     exit(4);
199                 }
200                 fprintf(stdout,"A selection has been made!\n");
201                 for(i = 0; i <= fdmax; i++){
202                         if (FD_ISSET(i, &read_fds)) { // we got one!!
203                         if (i == sd) {
204                             len = sizeof(struct sockaddr_in);
205                             // handle new connections
206                             fprintf(stderr, "Waiting for an incoming connection...\n");
207                             if ((newsd = accept(sd, (struct sockaddr *)&sa, &len)) < 0) {
208                                 perror("accept");
209                                 exit(1);
210                             }
211                             else{
212                                 FD_SET(newsd, &master); // add to master set
213                                 if (newsd > fdmax) {    // keep track of the max
214                                         fdmax = newsd;
215                                 }
216                                 peer[newsd].init=0;
217                                 peer[newsd].sd_link=newsd;
218                                 //Print everything is good;
219                                 if(!inet_ntop(AF_INET, &sa.sin_addr, addrstr, sizeof(addrstr))) {
220                                         perror("could not format IP address");
221                                         exit(1);
222                                 }
223                                 fprintf(stderr, "Incoming connection from %s:%d\n",addrstr, ntohs(
    sa.sin_port));
224                                 memset(&peer[i].address, 0, sizeof(peer[i].address));
225                                 sprintf(peer[newsd].address,"%s:%d",addrstr,ntohs(sa.sin_port));
226                                 /* Accept an incoming connection */
227                                 sprintf(A,"Welcome on board!\n");
228                                 if (insist_write(newsd, A, strlen(A)) == -1) {
229                                         perror("send");
230                                 }
231                                 /* Ask his name */
232                                 memset(&peer[i].name, 0, sizeof(peer[i].name));
233                                 peer[newsd].init = getname(&peer[newsd]);
234                                 if(peer[i].init==-1){
235                                         goto exit;
236                                 }
237                                 }
238                         }else{
239                                 if(peer[i].init==0){
240                                         peer[i].init = getname(&peer[i]);
241                                         if(peer[i].init==-1){
242                                                 goto exit;
243                                         }
244                                         continue;
```

```
                                }
                                n = read_msg(buf,i);
                                if (n <= 0) {
                                        if (n < 0){
                                                perror("read from remote peer failed");
                                        }
                                        else{
                                                fprintf(stderr, "Peer went away\n");
                                        }
exit:                                   if (close(i) < 0) perror("close");
                                        FD_CLR(i, &master);
                                        memset(&peer[i].name, 0, sizeof(peer[i].name));
                                        memset(&peer[i].address, 0, sizeof(peer[i].address));
                                        peer[i].init=-1;
                                }else{
                                        //toupper_buf(buf, n);
                                        sprintf(A,"User: %s with sd %d ::: with ip address : %s \
    nsays: %s\n",peer[i].name,peer[i].sd_link,peer[i].address,buf);
                                        for(j = 0; j <= fdmax; j++) {
                                        // send to everyone!
                                                if (FD_ISSET(j, &master)) {
                                                // except the listener and ourselves
                                                        if (j != sd && j != i) {
                                                                if (insist_write(j, A, strlen(A))
    == -1){
                                                                        perror("send");
                                                                }
                                                        }
                                                }
                                        }
                                }
                        }       // END handle data from client
                } // END got new incoming connection
            } // END looping through file descriptors
                        /* Make sure we don't leak open files */
        }// END for(;;)--and you thought it would never end!

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

```
        /*
        fprintf(stdout, "Client from %s:%d said : %s\n",addrstr, ntohs(sa.sin_port),buf);
                                if( send_msg(0,newsd,NULL) == -1){
                                        break;
                                }
        */
```

socket–client–z2.c

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

#include <stdio.h>
```

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

#include <sys/time.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/ioctl.h>
#include <sys/stat.h>

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


#include "socket-common.h"


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;

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;
}
/* 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 iv_n_key(int server_descriptor) {

        int i;
        ssize_t n;

        fprintf(stdout, "Receiving initialization vector ...\n");

        /* Read initialization vector */
        n = read(server_descriptor, data.iv, sizeof(data.iv));
        if (n < 0) {
                perror("read");
```

```
85                  exit(1);
86          }
87          if (n <= 0) return 1;
88
89          /* Print initialization vector */
90          fprintf(stdout, "Initialization vector received :\n");
91
92          for (i = 0; i < BLOCK_SIZE; i++) {
93                  printf("%x", data.iv[i]);
94          }
95          printf("\n");
96
97          fprintf(stdout, "Receiving symmetric key ...\n");
98
99          /* Read key */
100         n = read(server_descriptor, data.key, sizeof(data.key));
101         if (n < 0) {
102                 perror("read");
103                 exit(1);
104         }
105         if (n <= 0) return 1;
106
107         /* Print key */
108         fprintf(stdout, "Symmetric key received :\n");
109         for (i = 0; i < KEY_SIZE; i++) {
110                 printf("%x", data.key[i]);
111         }
112         printf("\n");
113
114         fprintf(stdout, "All done.\n");
115
116         return 0;
117
118 }
119
120
121 int encrypt_n_send(int crypto_fd, int server_descriptor) {
122
123         int i;
124
125         cryp.ses = sess.ses;
126         cryp.len = sizeof(data.in);
127         cryp.src = data.in;
128         cryp.dst = data.encrypted;
129         cryp.iv = data.iv;
130         cryp.op = COP_ENCRYPT;
131
132         fprintf(stdout, "Your PlainText Answer is:\n%s\n",data.in);
133
134         /* Encrypt data */
135         if (ioctl(crypto_fd, CIOCCRYPT, &cryp)) {
136                 perror("ioctl(CIOCCRYPT)");
137                 return 1;
138         }
139         /*
140         fprintf(stdout,"Your CipherText Answer is:\n");
141         for (i = 0; i < sizeof(data.encrypted); i++) {
142                 printf("%x", data.encrypted[i]);
143         }
144         printf("\n");
145         */
146         /* Send data */
147         if (insist_write(server_descriptor, data.encrypted, sizeof(data.encrypted)) != sizeof(data.
    encrypted)) {
148                 perror("write");
149                 exit(1);
150         }
151
152         return 0;
153 }
154 int decrypt_data(int crypto_fd) {
155
156         int i;
157         cryp.ses = sess.ses;
158         cryp.len = sizeof(data.in);
159         cryp.src = data.in;
```

```
160         cryp.src = data.encrypted;
161         cryp.dst = data.decrypted;
162         cryp.iv = data.iv;
163         cryp.op = COP_DECRYPT;
164         /*
165         printf("Encrypted Data from Server is:\n");
166         for (i = 0; i < DATA_SIZE; i++) {
167                 printf("%x", data.encrypted[i]);
168         }
169         printf("\n");
170         */
171         /* Decrypt data */
172         if (ioctl(crypto_fd, CIOCCRYPT, &cryp)) {
173                 perror("ioctl(CIOCCRYPT)");
174                 return 1;
175         }
176
177         fprintf(stdout, "Remote says:\n");
178         fprintf(stdout, "%s",data.decrypted);
179
180         return 0;
181 }
182
183 int main(int argc, char *argv[])
184 {
185
186         fd_set master;    // master file descriptor list
187         fd_set read_fds;  // temp file descriptor list for select()
188         int sd, port;
189         int i=0;
190         int fdmax;
191         int cfd;
192         int count;
193         char c;
194         ssize_t n;
195         char *hostname;
196         struct hostent *hp;
197         struct sockaddr_in sa;
198
199         /* Open dev crypto */
200         cfd = open("/dev/crypto", O_RDWR);
201         if (cfd < 0) {
202                 perror("open(/dev/crypto)");
203                 return 1;
204         }
205
206         memset(&sess, 0, sizeof(sess));
207         memset(&cryp, 0, sizeof(cryp));
208
209         if (argc != 3) {
210                 fprintf(stderr, "Usage: %s hostname port\n", argv[0]);
211                 exit(1);
212         }
213         hostname = argv[1];
214         port = atoi(argv[2]); /* Needs better error checking */
215
216
217         /* Create TCP/IP socket, used as main chat channel */
218         if ((sd = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
219                 perror("socket");
220                 exit(1);
221         }
222         fprintf(stderr, "Created TCP socket\n");
223
224
225         /* Look up remote hostname on DNS */
226         if ( !(hp = gethostbyname(hostname))) {
227                 printf("DNS lookup failed for host %s\n", hostname);
228                 exit(1);
229         }
230
231         /* Connect to remote TCP port */
232         sa.sin_family = AF_INET;
233         sa.sin_port = htons(port);
234         memcpy(&sa.sin_addr.s_addr, hp->h_addr, sizeof(struct in_addr));
235         fprintf(stderr, "Connecting to remote host... "); fflush(stderr);
```

```
236         if (connect(sd, (struct sockaddr *) &sa, sizeof(sa)) < 0) {
237                 perror("connect");
238                 exit(1);
239         }
240         fprintf(stderr, "Connected.\n");
241         /* Initialization vector and symmetric key */
242         if(iv_n_key(sd)) goto out;
243 
244         sess.cipher = CRYPTO_AES_CBC;
245         sess.keylen = KEY_SIZE;
246         sess.key = data.key;
247 
248         if (ioctl(cfd, CIOCGSESSION, &sess)) {
249                 perror("ioctl(CIOCGSESSION)");
250                 return 1;
251         }
252 
253         FD_ZERO(&master);    // clear the master and temp sets
254         FD_ZERO(&read_fds);
255         FD_SET(sd, &master);
256         FD_SET(0, &master);
257         fdmax = sd;
258         for(;;){
259                 read_fds = master; // copy it
260                 fprintf(stdout,"Waiting for a selection...\n");
261                 if (select(fdmax+1, &read_fds, NULL, NULL, NULL) == -1) {
262                     perror("select");
263                     exit(4);
264                 }
265                 fprintf(stdout,"A selection has been made!\n");
266                 if (FD_ISSET(sd, &read_fds)){
267                         /*
268                         * Let the remote know we're not going to write anything else.
269                         * Try removing the shutdown() call and see what happens.
270                         */
271                         memset(data.encrypted,0,sizeof(data.encrypted));
272                         /* Read answer and write it to standard output */
273                         n = read(sd, data.encrypted, sizeof(data.encrypted));
274                         if (n < 0) {
275                                 perror("read");
276                                 exit(1);
277                         }
278                         if (n <= 0){
279                                 goto out;
280                         }
281                         fprintf(stdout, "Server Replies:\n");
282                         memset(data.decrypted,0,sizeof(data.decrypted));
283                         if(decrypt_data(cfd)) return 1;
284                         //fprintf(stdout,"\n");
285                 }
286                 if (FD_ISSET(0, &read_fds)){
287                         i=0;
288                         memset(data.in,0,sizeof(data.in));
289                         fprintf(stdout,"Say Something to Server ::\n");
290                         /* Be careful with buffer overruns, ensure NUL-termination */
291                         //clearerr(stdin);
292 
293                         while((count = fscanf(stdin,"%c",&c))!=-1 && c!='\n' && i<=MESSAGE_SIZE-2){
294                                 data.in[i]=c;
295                                 i++;
296                         }
297                         if(count==-1) goto out;
298                         data.in[i] = '\0';
299 
300 
301                         /* Encrypt what you've said */
302                         if(encrypt_n_send(cfd,sd)) return 1;
303 
304                         /*
305                         * Let the remote know we're not going to write anything else.
306                         * Try removing the shutdown() call and see what happens.
307                         */
308                 }
309         }
310 out:
311         if (shutdown(sd, SHUT_WR) < 0) {
```

```
                        perror("shutdown");
                        exit(1);
        }
        if (ioctl(cfd, CIOCFSESSION, &sess.ses)) {
                perror("ioctl(CIOCFSESSION)");
                return 1;
        }
        if (close(cfd) < 0) {
                perror("close(fd)");
                return 1;
        }
        fprintf(stdout, "\nDone.\n");
        return 0;
}
```

socket–server–z2.c

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

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

#include <sys/ioctl.h>
#include <sys/time.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/stat.h>

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

#include <crypto/cryptodev.h>


#include "socket-common.h"

struct cry{
        unsigned char   in[DATA_SIZE],
                        encrypted[DATA_SIZE],
                        decrypted[DATA_SIZE],
                        iv[BLOCK_SIZE],
                        key[KEY_SIZE];
};
struct User{
        char name[100];
        char address[100];
        int sd_link;
        int init;
        struct cry data;
        struct session_op sess;
        struct crypt_op cryp;
};
struct User * peer;

void * safemalloc(size_t s){
        void *cd;
        cd = malloc(s);
        if(cd == NULL){
                perror("memory");
                exit(1);
        }
        return cd;
```

```
}
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;
}
/* 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 read_msg(char buf[MESSAGE_SIZE],int newsd){
        ssize_t n;
        char A[MESSAGE_SIZE];
        memset(A,0,sizeof(A));
        n = read(newsd, A, MESSAGE_SIZE*sizeof(char));
        //printf("READ MSG ::::: A == %s\n",A);
        strcpy(buf,A);
        return n;
}

int send_msg(int f,int newsd,char A[MESSAGE_SIZE]){
        /* if f equals 0 read from stdin */
        char ans[MESSAGE_SIZE];
        char c;
        int i;
        if(f == 0){
                printf("Reply ::\n");
                memset(ans,0,sizeof(ans));
                i=0;
                while(fscanf(stdin,"%c",&c)!=0 && c!='\n' && i<=MESSAGE_SIZE-2){
                                ans[i]=c;
                                i++;
                }
                ans[i] = '\0';
        }
        else{
                strcpy(ans,A);
        }
        fprintf(stdout, "Your Answer is:\n%s\n",ans);
        if (insist_write(newsd,ans,strlen(ans)) != strlen(ans)) {
                        perror("write to remote peer failed");
                        return -1;
        }
        return 0;
}

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)
```

```
134                 return crypto_fd;
135
136         ret = insist_read(crypto_fd, buf, cnt);
137         close(crypto_fd);
138
139         return ret;
140 }
141
142 int iv_n_key_server(int new_sd) {
143         int i;
144         /* Create and print initialization vector */
145         fprintf(stdout, "Creating initialization vector ...\n");
146         if (fill_urandom_buf(peer[new_sd].data.iv, BLOCK_SIZE) < 0) {
147                 perror("getting data from /dev/urandom\n");
148                 return 1;
149         }
150         fprintf(stdout, "Initialization vector created :\n");
151         for (i = 0; i < BLOCK_SIZE; i++) {
152                 printf("%x", peer[new_sd].data.iv[i]);
153         }
154         printf("\n");
155
156         /* Send initialization vector to client */
157         if (insist_write(new_sd,peer[new_sd].data.iv,sizeof(peer[new_sd].data.iv)) != sizeof(peer[
    new_sd].data.iv)) {
158                 perror("write to remote peer failed");
159                 return 2;
160         }
161
162         /* Create and print symmetric key */
163         fprintf(stdout, "Creating symmetric key ...\n");
164         if (fill_urandom_buf(peer[new_sd].data.key, KEY_SIZE) < 0) {
165                 perror("getting data from /dev/urandom\n");
166                 return 1;
167         }
168         fprintf(stdout, "Symmetric key created :\n");
169         for (i = 0; i < KEY_SIZE; i++) {
170                 printf("%x", peer[new_sd].data.key[i]);
171         }
172         printf("\n");
173
174         /* Send symmetric key to client */
175         if (insist_write(new_sd,peer[new_sd].data.key,sizeof(peer[new_sd].data.key)) != sizeof(peer
    [new_sd].data.key)) {
176                 perror("write to remote peer failed");
177                 return 2;
178         }
179
180         return 0;
181 }
182 int encrypt_n_send_to_client(int crypto_fd, int new_sd){
183
184         int i;
185         peer[new_sd].cryp.ses = peer[new_sd].sess.ses;
186         peer[new_sd].cryp.len = sizeof(peer[new_sd].data.in);
187         peer[new_sd].cryp.src = peer[new_sd].data.in;
188         peer[new_sd].cryp.dst = peer[new_sd].data.encrypted;
189         peer[new_sd].cryp.iv = peer[new_sd].data.iv;
190         peer[new_sd].cryp.op = COP_ENCRYPT;
191
192         /*
193         fprintf(stdout, "Your PlainText Answer is:\n%s\n",peer[new_sd].data.in);
194         */
195         /* Encrypt data */
196         if (ioctl(crypto_fd, CIOCCRYPT, &peer[new_sd].cryp)) {
197                 perror("ioctl(CIOCCRYPT)");
198                 return 1;
199         }
200         /*
201         fprintf(stdout,"Your CipherText Answer is:\n");
202         for (i = 0; i < sizeof(peer[new_sd].data.encrypted); i++) {
203                 printf("%x", peer[new_sd].data.encrypted[i]);
204         }
205         printf("\n");
206         */
207         /* Send data */
```

```
208         if (insist_write(new_sd,peer[new_sd].data.encrypted,sizeof(peer[new_sd].data.encrypted)) !=
      sizeof(peer[new_sd].data.encrypted)) {
209                 perror("write to remote peer failed");
210                 return 2;
211         }
212
213         return 0;
214 }
215 int decrypt_data(int sd, int crypto_fd){
216         int i;
217         peer[sd].cryp.ses = peer[sd].sess.ses;
218         peer[sd].cryp.len = sizeof(peer[sd].data.in);
219         peer[sd].cryp.src = peer[sd].data.in;
220         peer[sd].cryp.src = peer[sd].data.encrypted;
221         peer[sd].cryp.dst = peer[sd].data.decrypted;
222         peer[sd].cryp.iv = peer[sd].data.iv;
223         peer[sd].cryp.op = COP_DECRYPT;
224
225         /*
226         printf("Encrypted Data from Client is:\n");
227         for (i = 0; i < DATA_SIZE; i++) {
228                 printf("%x", peer[sd].data.encrypted[i]);
229         }
230         printf("\n");
231         */
232         /* Decrypt data */
233         if (ioctl(crypto_fd, CIOCCRYPT, &peer[sd].cryp)) {
234                 perror("ioctl(CIOCCRYPT)");
235                 return 1;
236         }
237         return 0;
238 }
239 int getname(int sd, int crypto_fd){
240         ssize_t n;
241         int retval;
242         struct timeval tv;
243
244         fd_set master;    // master file descriptor list
245         fd_set read_fds;  // temp file descriptor list for select()
246         FD_ZERO(&master);    // clear the master and temp sets
247         FD_ZERO(&read_fds);
248         FD_SET(sd, &master);
249         tv.tv_sec = WAIT_SEC;
250         tv.tv_usec = 0;
251         sprintf(peer[sd].data.in,"Who are you?\n");
252         encrypt_n_send_to_client(crypto_fd,sd);
253         read_fds = master; // copy it
254         fprintf(stdout,"Waiting for a selection... In ... getname()\n");
255         retval = select(sd+1, &read_fds, NULL, NULL, &tv);
256         if(retval == -1){
257                 perror("select");
258                 return -1;
259         }
260         else{
261                 if(retval){
262                         n = read(sd,peer[sd].data.encrypted, sizeof(peer[sd].data.encrypted));
263                         if (n <= 0) {
264                                 if (n < 0){
265                                         perror("read from remote peer failed");
266                                 }
267                                 else{
268                                         fprintf(stderr, "Peer went away\n");
269                                 }
270                                 return -1;
271                         }
272                         fprintf(stdout,"A selection has been made..\n");
273                         decrypt_data(sd,crypto_fd);
274                         sprintf(peer[sd].name,"%s",peer[sd].data.decrypted);
275                         return 1;
276                 }
277                 else{
278                         printf("Peer with sd : %d on address %s didn't answer get_name question
    within %d seconds.\n",peer[sd].sd_link,peer[sd].address,WAIT_SEC);
279                         return 0;
280                 }
281         }
```

```
}
int main(void)
{
        fd_set master;    // master file descriptor list
        fd_set read_fds;  // temp file descriptor list for select()
        int fdmax;        // maximum file descriptor number
        //char A[MESSAGE_SIZE+200];
        char addrstr[INET_ADDRSTRLEN];
        int sd, newsd;
        int cfd;
        peer = (struct User *)safemalloc((PEER_LEN)*sizeof(struct User));

        int i,j;
        int x;
        ssize_t n;
        socklen_t len;
        struct sockaddr_in sa;

        /* Make sure a broken connection doesn't kill us */
        signal(SIGPIPE, SIG_IGN);
        FD_ZERO(&master);    // clear the master and temp sets
        FD_ZERO(&read_fds);

        /* Open dev crypto */
        cfd = open("/dev/crypto", O_RDWR);
        if (cfd < 0) {
                perror("open(/dev/crypto)");
                return 1;
        }


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

        /* 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);
        }
        // add the listener to the master set
        FD_SET(sd, &master);

        // keep track of the biggest file descriptor
        fdmax = sd; // so far, it's this one
        /* Loop forever, accept()ing connections */
        for (;;) {
                read_fds = master; // copy it
                fprintf(stdout,"Waiting for a selection...\n");
                if (select(fdmax+1, &read_fds, NULL, NULL, NULL) == -1) {
                    perror("select");
                    exit(4);
                }
                fprintf(stdout,"A selection has been made!\n");
                for(i = 0; i <= fdmax; i++){
                        //printf("\n\nIf case 0! i = %d \n\n",i);
                        if (FD_ISSET(i, &read_fds)) { // we got one!!
                                if (i == sd){
                                   //printf("\n\nIf case 1! \n\n");
                                    len = sizeof(struct sockaddr_in);
                                    // handle new connections
                                    fprintf(stderr, "Waiting for an incoming connection...\n");
```

```
358                                     if ((newsd = accept(sd, (struct sockaddr *)&sa, &len)) < 0) {
359                                         perror("accept");
360                                         exit(1);
361                                     }
362                                     else{
363                                         printf("\n\nIf case 2! \n\n");
364                                         FD_SET(newsd, &master); // add to master set
365                                         if (newsd > fdmax) {    // keep track of the max
366                                                 fdmax = newsd;
367                                         }
368                                         peer[newsd].init=0;
369                                         peer[newsd].sd_link=newsd;
370                                         //Print everything is good;
371                                         if(!inet_ntop(AF_INET, &sa.sin_addr, addrstr, sizeof(
    addrstr))) {
372                                                 perror("could not format IP address");
373                                                 exit(1);
374                                         }
375                                         fprintf(stdout, "Incoming connection from %s:%d\n",addrstr,
     ntohs(sa.sin_port));
376                                         sprintf(peer[newsd].address,"%s:%d",addrstr,ntohs(sa.
    sin_port));
377                                         x=iv_n_key_server(newsd);
378                                         if (x==1) return 1;
379                                         else if (x==2) break;
380                                         /* Initialise Crypto */
381                                         peer[newsd].sess.cipher = CRYPTO_AES_CBC;
382                                         peer[newsd].sess.keylen = KEY_SIZE;
383                                         peer[newsd].sess.key = peer[newsd].data.key;
384
385                                         if (ioctl(cfd, CIOCGSESSION, &peer[newsd].sess)){
386                                                 perror("ioctl(CIOCGSESSION)");
387                                                 return 1;
388                                         }
389                                         /* Accept an incoming connection */
390                                         sprintf(peer[newsd].data.in,"Welcome on board!\n");
391                                         x=encrypt_n_send_to_client(cfd,newsd);
392                                         /* Ask his name */
393                                         if (x==1) return 1;
394                                         else if (x==2) break;
395                                         peer[newsd].init = getname(newsd,cfd);
396                                                 if(peer[newsd].init==-1){
397                                                         goto exit;
398                                                 }
399                                         }
400                                 }else{
401                                         //printf("\n\nIf case 3! \n\n");
402                                         if(peer[i].init==0){
403                                                 peer[i].init = getname(i,cfd);
404                                                 if(peer[i].init==-1){
405                                                         //printf("\n\nIf case 3 exit! \n\n");
406                                                         goto exit;
407                                                 }
408                                                 continue;
409                                         }
410                                         //printf("\n\nIf case 3 .. 011! \n\n");
411                                         n = read(i, peer[i].data.encrypted, sizeof(peer[i].data.
    encrypted));
412                                         //printf("\n\nIf case 3 .. 1! \n\n");
413                                         if (n <= 0) {
414                                                 if (n < 0){
415                                                 perror("read from remote peer failed");
416                                                 }
417                                                 else{
418                                                         fprintf(stderr, "Peer went away\n");
419                                                         //printf("\n\nIf case 3 .. 2! \n\n");
420
421 exit:                                           }
422                                                 if (close(i) < 0) perror("close");
423                                                 FD_CLR(i, &master);
424                                                 if (ioctl(cfd, CIOCFSESSION, &peer[i].sess.ses)) {
425                                                         perror("ioctl(CIOCFSESSION)");
426                                                         return 1;
427                                                 }
428                                                 peer[i].init=-1;
429                                         }else{
                                                        //printf("\n\nIf case 4! \n\n");
```

```
                                                if(decrypt_data(i,cfd)) return 1;
                                                sprintf(peer[i].data.in,"User: %s ::: with sd %d ::: with ip address : %s \nsays: %s\n",peer[i].name,peer[i].sd_link,peer[i].address,peer[i].data.decrypted);
                                                for(j = 0; j <= fdmax; j++) {
                                                // send to everyone!
                                                        if (FD_ISSET(j, &master)){
                                                        // except the listener and ourselves
                                                                if (j != sd && j != i) {
                                                                        //printf("\n\nIf case 5! \n\n");
                                                                        sprintf(peer[j].data.in,"%s",peer[i].data.in);
                                                                        printf("Peer with sd = %d sending to peer with sd = %d, from cfd = %d the text %s",i,j,cfd,peer[i].data.in);
                                                                        x=encrypt_n_send_to_client(cfd,j);
                                                                        if (x==1) return 1;
                                                                        else if (x==2) break;
                                                                }
                                                        }
                                                }
                                        }
                                }     // END handle data from client
                        } // END got new incoming connection
                } // END looping through file descriptors
                        /* Make sure we don't leak open files */
        }// END for(;;)--and you thought it would never end!
        /* This will never happen */
        return 1;
}
```

```
        /*
        fprintf(stdout, "Client from %s:%d said : %s\n",addrstr, ntohs(sa.sin_port),buf);
                                if( send_msg(0,newsd,NULL) == -1){
                                        break;
                                }
        */
```

crypto–chrdev.c

```
/*
 * 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/poll.h>
#include <linux/sched.h>
#include <linux/module.h>
#include <linux/wait.h>
#include <linux/virtio.h>
```

```c
#include <linux/virtio_config.h>

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

#include "cryptodev.h"

#define IV_SIZE 16
/*
 * Global data
 */
struct cdev crypto_chrdev_cdev;

/**
 * Given the minor number of the inode return the crypto device
 * that owns that number.
 **/

static struct crypto_device *get_crypto_dev_by_minor(unsigned int minor)
{
        struct crypto_device *crdev;
        unsigned long flags;

        debug("Entering");

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

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

        debug("Leaving");
        return crdev;
}

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

static int crypto_chrdev_open(struct inode *inode, struct file *filp)
{
        int ret = 0;
        int err;
        unsigned int len;
        struct crypto_open_file *crof;
        struct crypto_device *crdev;
        unsigned int syscall_type = VIRTIO_CRYPTO_SYSCALL_OPEN;
        int host_fd = -1;
        unsigned long flag;
        struct scatterlist *sglist[2], syscall_type_sg, host_fd_sg;

        debug("Entering");

        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;
```

```
94          }
95          crof->crdev = crdev;
96          crof->host_fd = -1;
97          debug("SYSCALL_TYPE = %d",syscall_type);
98          /**
99           * We need two sg lists, one for syscall_type and one to get the
100          * file descriptor from the host.
101          **/
102
103
104
105         sg_init_one(&syscall_type_sg, &syscall_type, sizeof(syscall_type));
106         sglist[0]=&syscall_type_sg;
107         sg_init_one(&host_fd_sg, &host_fd, sizeof(host_fd));
108         sglist[1]=&host_fd_sg;
109         debug("Created sg_lists: launching vitqueue_add");
110         spin_lock_irqsave(&crdev->vq_lock,flag);
111         err = virtqueue_add_sgs(crdev->vq,sglist,1,1,&syscall_type_sg,GFP_ATOMIC);
112         debug("Kicking");
113         virtqueue_kick(crdev->vq);
114         /**
115          * Wait for the host to process our data.
116         **/
117         debug("virtqueue_get_buf");
118         while(virtqueue_get_buf(crdev->vq,&len)==NULL);
119         spin_unlock_irqrestore(&crdev->vq_lock,flag);
120         crof->host_fd = host_fd;
121         debug("len = %d",len);
122         debug("SYSCALL_TYPE = %d",syscall_type);
123         debug("host_fd = %d",host_fd);
124         debug("ret = %d",ret);
125         if(host_fd == -1){
126                 ret = -ENODEV;
127                 goto fail;
128         }
129         filp->private_data = crof;
130
131         /*
132          If host failed to open() return -ENODEV. */
133
134
135 fail:
136         debug("Leaving");
137         return ret;
138 }
139
140 static int crypto_chrdev_release(struct inode *inode, struct file *filp)
141 {
142         int ret = 0;
143         int err = 0;
144         int len = 0;
145         struct crypto_open_file *crof = filp->private_data;
146         struct crypto_device *crdev = crof->crdev;
147         unsigned int syscall_type = VIRTIO_CRYPTO_SYSCALL_CLOSE;
148         struct scatterlist *sglist[3], syscall_type_sg, host_fd_sg, syscall_ret;
149         int host_fd = crof->host_fd;
150         unsigned long flag;
151
152         sg_init_one(&syscall_type_sg, &syscall_type, sizeof(syscall_type));
153         sglist[0]=&syscall_type_sg;
154         sg_init_one(&host_fd_sg, &host_fd, sizeof(host_fd));
155         sglist[1]=&host_fd_sg;
156         sg_init_one(&syscall_ret,&ret,sizeof(ret));
157         sglist[2]=&syscall_ret;
158         debug("Entering");
159
160         /**
161          * Send data to the host.
162          **/
163         spin_lock_irqsave(&crdev->vq_lock,flag);
164         err = virtqueue_add_sgs(crdev->vq,sglist,2,1,&syscall_type_sg,GFP_ATOMIC);
165         debug("Kicking");
166         virtqueue_kick(crdev->vq);
167
168         /**
169          * Wait for the host to process our data.
```

```
170             **/
171             debug("virtqueue_get_buf");
172             while(virtqueue_get_buf(crdev->vq,&len)==NULL);
173             spin_unlock_irqrestore(&crdev->vq_lock,flag);
174             debug("close returned ret = %d",ret);
175 
176             kfree(crof);
177             debug("Leaving");
178             return ret;
179 
180     }
181 
182     static long crypto_chrdev_ioctl(struct file *filp, unsigned int cmd, unsigned long arg)
183     {
184             struct session_op sess;
185             struct crypt_op cryp;
186             long ret = 0;
187             int i=0;
188             int err;
189             unsigned int md = cmd;
190             struct crypto_open_file *crof = filp->private_data;
191             struct crypto_device *crdev = crof->crdev;
192             struct virtqueue *vq = crdev->vq;
193             int fd = crof->host_fd;
194             unsigned char *session_key;
195             unsigned char *src;
196             unsigned char *iv;
197             unsigned char *dst=NULL;
198             unsigned long flag;
199             struct scatterlist syscall_type_sg,sess_arg, src_sg, iv_sg, dest_sg, arg_sg, fd_sg, type_sg
        , ret_sg, *sgs[8];
200     #define MSG_LEN 100
201             unsigned char output_msg[MSG_LEN], input_msg[MSG_LEN];
202             unsigned int num_out, num_in,syscall_type = VIRTIO_CRYPTO_SYSCALL_IOCTL,len;
203             debug("Entering");
204 
205             num_out = 0;
206             num_in = 0;
207 
208             /**
209              *  These are common to all ioctl commands.
210              **/
211             sg_init_one(&syscall_type_sg, &syscall_type, sizeof(syscall_type));
212             sgs[num_out++] = &syscall_type_sg;
213             /* ?? */
214             sg_init_one(&fd_sg,&fd,sizeof(fd));
215             sgs[num_out++] = &fd_sg;
216             sg_init_one(&type_sg, &md, sizeof(md));
217             sgs[num_out++] = &type_sg;
218             /**
219              *  Add all the cmd specific sg lists.
220              **/
221             debug("fd = %d",fd);
222             debug("CIOCGSESSION = %ld",CIOCGSESSION);
223             debug("CIOCFSESSION = %ld",CIOCFSESSION);
224             debug("CIOCCRYPT = %ld",CIOCCRYPT);
225             debug("cmd = %d",md);
226             switch (cmd) {
227             case CIOCGSESSION:
228                     debug("CIOCGSESSION");
229                     err = copy_from_user(&sess,(struct session_op __user *)arg,sizeof(sess));
230                     debug("CRYPTO_AES_CBC = %d",CRYPTO_AES_CBC);
231                     debug("sess.cipher = %d",sess.cipher);
232                     //debug("KEY_SIZE = %d",KEY_SIZE);
233                     debug("sess.keylen = %d",sess.keylen);
234 
235 
236                     session_key = kmalloc(sess.keylen*sizeof(unsigned char),GFP_KERNEL);
237                     memset(session_key,0,sess.keylen*sizeof(unsigned char));
238                     err = copy_from_user(session_key,((struct session_op __user *)arg)->key,sess.keylen
    *sizeof(unsigned char));
239                     sg_init_one(&sess_arg, &session_key,sess.keylen*sizeof(unsigned char));
240                     sgs[num_out++]=&sess_arg;
241 
242 
243                     sg_init_one(&arg_sg, &sess,sizeof(sess));
```

```
244                 sgs[num_out + num_in++] = &arg_sg;
245
246                 break;
247
248         case CIOCFSESSION:
249                 debug("CIOCFSESSION");
250                 err = copy_from_user(&sess,(struct session_op __user *)arg,sizeof(sess));
251                 sg_init_one(&arg_sg, &sess,sizeof(sess));
252                 sgs[num_out++] = &arg_sg;
253                 break;
254
255         case CIOCCRYPT:
256                 debug("CIOCCRYPT");
257                 err = copy_from_user(&cryp,(struct crypt_op __user *)arg,sizeof(cryp));
258                 sg_init_one(&arg_sg, &cryp,sizeof(cryp));
259                 sgs[num_out++] = &arg_sg;
260
261
262                 src = kmalloc(cryp.len*sizeof(unsigned char),GFP_KERNEL);
263                 memset(src,0,cryp.len*sizeof(unsigned char));
264                 err = copy_from_user(src,((struct crypt_op __user *)arg)->src,cryp.len*sizeof(
    unsigned char));
265                 sg_init_one(&src_sg, src,cryp.len*sizeof(unsigned char));
266                 sgs[num_out++] = &src_sg;
267
268
269                 iv = kmalloc(IV_SIZE*sizeof(unsigned char),GFP_KERNEL);
270                 memset(iv,0,IV_SIZE*sizeof(unsigned char));
271                 err = copy_from_user(iv,((struct crypt_op __user *)arg)->iv,IV_SIZE*sizeof(unsigned
     char));
272                 sg_init_one(&iv_sg, iv,IV_SIZE*sizeof(unsigned char));
273                 sgs[num_out++] = &iv_sg;
274
275                 dst = kmalloc(cryp.len*sizeof(unsigned char),GFP_KERNEL);
276                 memset(dst,0,cryp.len*sizeof(unsigned char));
277                 sg_init_one(&dest_sg, dst,cryp.len*sizeof(unsigned char));
278                 sgs[num_out + num_in++] = &dest_sg;
279
280                 debug("cryp.len = %d",cryp.len);
281                 for (i = 0; i < cryp.len; i++) {
282                         printk(KERN_DEBUG "%x", cryp.src[i]);
283                 }
284
285                 break;
286         default:
287                 debug("Unsupported ioctl command");
288                 break;
289         }
290         sg_init_one(&ret_sg, &ret,sizeof(ret));
291         sgs[num_out + num_in++] = &ret_sg;
292         /**
293          * Wait for the host to process our data.
294          **/
295         /* ?? */
296         /* ?? Lock ?? */
297         debug("Virtqueue add sgs on ioctl::");
298         debug("num_out = %d",num_out);
299         debug("num_in = %d",num_in);
300         spin_lock_irqsave(&crdev->vq_lock,flag);
301         err = virtqueue_add_sgs(crdev->vq, sgs, num_out, num_in, &syscall_type_sg, GFP_ATOMIC);
302         virtqueue_kick(crdev->vq);
303         while (virtqueue_get_buf(crdev->vq, &len) == NULL);
304         /* do nothing */
305         spin_unlock_irqrestore(&crdev->vq_lock,flag);
306         switch (cmd) {
307         case CIOCGSESSION:
308         case CIOCFSESSION:
309                 debug("copy_to_user:  CIOCG/FSESSION");
310                 err = copy_to_user((struct session_op __user *)arg,&sess,sizeof(sess));
311                 break;
312         case CIOCCRYPT:
313                 debug("copy_to_user:  CIOCCRYPT");
314                 err = copy_to_user(((struct crypt_op __user *)arg)->dst,dst,cryp.len*sizeof(
    unsigned char));
315                 break;
316         default:
```

```
317                 debug("Unsupported ioctl command");
318                 break;
319         }
320
321         debug("<troika>");
322
323         debug("Leaving with ret = %ld",ret);
324
325         return ret;
326 }
327
328 static ssize_t crypto_chrdev_read(struct file *filp, char __user *usrbuf,
329                                   size_t cnt, loff_t *f_pos)
330 {
331         debug("Entering");
332         debug("Leaving");
333         return -EINVAL;
334 }
335
336 static struct file_operations crypto_chrdev_fops =
337 {
338         .owner          = THIS_MODULE,
339         .open           = crypto_chrdev_open,
340         .release        = crypto_chrdev_release,
341         .read           = crypto_chrdev_read,
342         .unlocked_ioctl = crypto_chrdev_ioctl,
343 };
344
345 int crypto_chrdev_init(void)
346 {
347         int ret;
348         dev_t dev_no;
349         unsigned int crypto_minor_cnt = CRYPTO_NR_DEVICES;
350
351         debug("Initializing character device...");
352         cdev_init(&crypto_chrdev_cdev, &crypto_chrdev_fops);
353         crypto_chrdev_cdev.owner = THIS_MODULE;
354
355         dev_no = MKDEV(CRYPTO_CHRDEV_MAJOR, 0);
356         ret = register_chrdev_region(dev_no, crypto_minor_cnt, "crypto_devs");
357         if (ret < 0) {
358                 debug("failed to register region, ret = %d", ret);
359                 goto out;
360         }
361         ret = cdev_add(&crypto_chrdev_cdev, dev_no, crypto_minor_cnt);
362         if (ret < 0) {
363                 debug("failed to add character device");
364                 goto out_with_chrdev_region;
365         }
366
367         debug("Completed successfully");
368         return 0;
369
370 out_with_chrdev_region:
371         unregister_chrdev_region(dev_no, crypto_minor_cnt);
372 out:
373         return ret;
374 }
375
376 void crypto_chrdev_destroy(void)
377 {
378         dev_t dev_no;
379         unsigned int crypto_minor_cnt = CRYPTO_NR_DEVICES;
380
381         debug("entering");
382         dev_no = MKDEV(CRYPTO_CHRDEV_MAJOR, 0);
383         cdev_del(&crypto_chrdev_cdev);
384         unregister_chrdev_region(dev_no, crypto_minor_cnt);
385         debug("leaving");
386 }
```

code_segment_from_qemu_virtio.c

```
1 static void vq_handle_output(VirtIODevice *vdev, VirtQueue *vq)
```

```
{
        VirtQueueElement elem;
        unsigned int *syscall_type;
        struct session_op *sess;
        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");
                int *user_fd =  elem.in_sg[0].iov_base;
                *user_fd = -1;
                printf("elem.in_sg[0].iov_base = %d before change\n",*((int *)(elem.in_sg[0].
    iov_base)));
                *(user_fd) = open(CRYPTODEV_FILENAME, O_RDWR);
                DEBUG("OPENING DEV CRYPTO");
                printf("USER_FD = %d\n",*user_fd);
                printf("elem.in_sg[0].iov_base = %d\n",*((int *)(elem.in_sg[0].iov_base)));
                break;

        case VIRTIO_CRYPTO_SYSCALL_TYPE_CLOSE:
                DEBUG("VIRTIO_CRYPTO_SYSCALL_TYPE_CLOSE");
                int *userfd = elem.out_sg[1].iov_base;
                int *ret = elem.in_sg[0].iov_base;
                *(ret)=close(*userfd);
                DEBUG("Close : ");
                printf("file with user_fd = %d and close() returned %d\n",*userfd,*ret);
                if(*(ret)==-1){
                        perror("close");
                }
                break;

        case VIRTIO_CRYPTO_SYSCALL_TYPE_IOCTL:
                DEBUG("VIRTIO_CRYPTO_SYSCALL_TYPE_IOCTL");
                int *hfd = elem.out_sg[1].iov_base;
                unsigned int *cmd = elem.out_sg[2].iov_base;
                long *cret;
                printf("CIOCGSESSION = %ld\n",CIOCGSESSION);
                printf("CIOCFSESSION = %ld\n",CIOCFSESSION);
                printf("CIOCCRYPT = %ld\n",CIOCCRYPT);
                printf("FD = %d\n",*hfd);
                printf("CMD = %d\n",*cmd);
                switch (*cmd) {
                case CIOCGSESSION:
                        DEBUG("CIOCGSESSION");
                        sess = (struct session_op *) elem.in_sg[0].iov_base;
                        sess->key = (unsigned char *) elem.out_sg[3].iov_base;
                        cret = elem.in_sg[1].iov_base;
                        printf("CRYPTO_AES_CBC = %d\n",CRYPTO_AES_CBC);
                        printf("sess.cipher = %d\n",sess->cipher);
                        //debug("KEY_SIZE = %d",KEY_SIZE);
                        printf("sess.keylen = %d\n",sess->keylen);
                        printf("ret of ioctl = %ld\n",*cret);
                        *cret=ioctl(*hfd, CIOCGSESSION, sess);
                        if (*cret) {
                                perror("ioctl(CIOCGSESSION)");
                                //return 1;
                        }
                        break;
                case CIOCFSESSION:
                        DEBUG("CIOCFSESSION");
                        sess = (struct session_op *) elem.out_sg[3].iov_base;
                        //struct session_op sss;
                        cret = elem.in_sg[0].iov_base;
                        printf("CRYPTO_AES_CBC = %d\n",CRYPTO_AES_CBC);
                        printf("sess.cipher = %d\n",sess->cipher);
                        //debug("KEY_SIZE = %d",KEY_SIZE);
                        printf("sess.keylen = %d\n",sess->keylen);
                        printf("ret of ioctl = %ld\n",*cret);
```

```
77                          //memcpy(&sss, sess, sizeof(sss));
78                          *cret=ioctl(*hfd, CIOCFSESSION, sess);
79                          if (*cret) {
80                                  perror("ioctl(CIOCFSESSION)");
81                                  //return 1;
82                          }
83                          //memcpy(sess,&sss,sizeof(sss));
84                          break;
85                  case CIOCCRYPT:
86                          DEBUG("CIOCCRYPT");
87                          struct crypt_op *cryp = (struct crypt_op *)elem.out_sg[3].iov_base;
88                          cryp->src = (unsigned char *)elem.out_sg[4].iov_base;
89                          cryp->iv = (unsigned char *)elem.out_sg[5].iov_base;
90                          cryp->dst = (unsigned char *)elem.in_sg[0].iov_base;
91                          printf("cryp.len = %d ",cryp->len);
92
93                          cret = elem.in_sg[1].iov_base;
94                          *cret=ioctl(*hfd, CIOCCRYPT, cryp);
95                          if (*cret) {
96                                  perror("ioctl(CIOCCRYPT)");
97                                  //return 1;
98                          }
99                          break;
100                 default:
101                         DEBUG("Unsupported ioctl command");
102                         break;
103                 }
104                 //memcpy(input_msg, "Host: Welcome to the virtio World!", 35);
105                 //printf("Guest says: %s\n", output_msg);
106                 //printf("We say: %s\n", input_msg);
107
108                 break;
109
110         default:
111                 DEBUG("Unknown syscall_type");
112         }
113
114         virtqueue_push(vq, &elem, 0);
115         virtio_notify(vdev, vq);
116 }
```