Εργαστήριο Λειτουργικών Συστημάτων

Αναφορά 3ης Εργαστηριακής Άσκησης

Φίλιππος Μαλανδράκης 03116200
Βιτάλιος Σαλής 03115751

Εισαγωγή

Σε αυτή την εργαστηριακή άσκηση μας ζητείται να υλοποιήσουμε:

1. Ένα chat πάνω απο TCP/IP sockets, το οποίο
2. χρησιμοποιεί end-to-end encryption, δηλαδή οι συνομιλητές μοιράζονται ένα κλειδί το οποίο χρησιμοποιούν για την κρυπτογράφηση/αποκρυπτογράφηση μηνυμάτων.
3. Μια εικονική συσκευή cryptodev, χρησιμοποιώντας ως βάση το VirtIO έτσι ώστε η κρυπτογράφηση χρησιμοποιώντας κρυπτογραφικούς επιταχυντές στο υλικό να είναι δυνατή και εντός της εικονικής μηχανής.

1 - Chat Over TCP/IP Using Sockets

Σε αυτή την ενότητα θα περιγράψουμε την υλοποίηση ενός chat room χρησιμοποιώντας sockets. Συγκεκριμένα, υλοποιούμε έναν server ο οποίος ακούει σε μια θύρα στο μηχάνημα και διάφοροι clients έχουν την δυνατότητα να συνδεθούν και να επικοινωνήσουν μεταξύ τους.

Για την υλοποίηση του server, αρχικά ζητάμε απο το λειτουργικό ένα listening socket μέσω της ακόλουθης μεθόδου:

```
int get_listener_socket(void) {
        int listener;
        int yes = 1;
        int rv;
        struct addrinfo hints, *ai, *p;

        memset(&hints, 0, sizeof hints);
        hints.ai_family = AF_UNSPEC; // don't care whether IPv4 or IPv6
        hints.ai_socktype = SOCK_STREAM; // stream socket
        hints.ai_flags = AI_PASSIVE; // get the host's address

        if ((rv = getaddrinfo(NULL, PORT, &hints, &ai)) == -1) {
                fprintf(stderr, "couldn't get address info %s", gai_strerror(rv));
                exit(1);
        }

        for (p = ai; p != NULL; p = p->ai_next) {
                listener = socket(p->ai_family, p->ai_socktype, p->ai_protocol);
                if (listener == -1) {
                    perror("socket");
                    continue;
                }

                setsockopt(listener, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int));

                if (bind(listener, p->ai_addr, p->ai_addrlen) < 0) {
                        perror("bind");
                        close(listener);
                        Continue;
                }
                break;
        }
        if (p == NULL) {
                return -1;
        }
        freeaddrinfo(ai);
        if (listen(listener, MAXCONN) == -1) {
                perror("listen");
                close(listener);
                return -1;
        }

        return listener;
}
```

Αρχικά, λαμβάνουμε τις πιθανές διευθύνσεις που μπορούμε να χρησιμοποιήσουμε. Στη

συνέχεια, για κάθε μια απο αυτές δοκιμάζουμε αμα μπορούμε να δημιουργήσουμε socket και να κάνουμε bind αυτό το socket στην διεύθυνση. Όταν βρούμε ένα socket που μπορεί να κάνει αυτές τις λειτουργίες κάνουμε listen σε αυτό και επιστρέφουμε τον file descriptor του.

Αφού ο server κάνει listen σε μια διεύθυνση, χρειάζεται να “ακούει” για νέες συνδέσεις απο clients. Ταυτόχρονα, πρέπει να παρακολουθεί μήπως οι υπάρχοντες client έχουν στείλει ένα μήνυμα ώστε να το προωθήσει στους υπόλοιπους.

Αυτές οι ανάγκες μας οδηγούν στο να χρησιμοποιήσουμε την δομή *pollfd*. Η δομή αυτή είναι ένας πίνακας απο file descriptors τους οποίους συσχετιζουμε είτε με είσοδο δεδομένων (*POLLIN*), είτε με έξοδο δεδομένων (*POLLOUT*). Μιας και ο server λαμβάνει συνδέσεις και δεδομένα, συσχετιζουμε όλους τους file descriptor με είσοδο (*POLLIN*). Μέσω του system call *poll* μπορούμε να κάνουμε query αυτούς τους file descriptor και να δούμε ποιοι απο αυτούς περιμένουν να μας δώσουν δεδομένα.

Μιας και στο chat μπορεί να συνδεθεί απροσδιόριστος αριθμός απο clients, χρησιμοποιούμε τις δυο ακόλουθες συναρτήσεις για την δυναμική διαχείριση του πίνακα *pollfd*:

```
void add_to_pfds(struct pollfd **pfds, int newfd, int *fd_count, int *fd_size) {
        if (*fd_count == *fd_size) {
                *fd_size *= 2;
                *pfds = realloc(*pfds, sizeof(**pfds) * *fd_size);
        }

        (*pfds)[*fd_count].fd = newfd;
        (*pfds)[*fd_count].events = POLLIN;

        (*fd_count)++;
}

void del_from_pfds(struct pollfd **pfds, int i, int *fd_count) {
        int n = *fd_count;
        (*pfds)[i] = (*pfds)[n - 1];
        *fd_count -= 1;
}
```

Η *add_to_pfds* λαμβάνει έναν νέο file descriptor, τον υπάρχοντα αριθμό απο file descriptors και το συνολικό μέγεθος του πίνακα. Αμα το νέο στοιχείο δεν χωράει, τότε κάνουμε realloc και διπλασιάζουμε το μέγεθος του πίνακα. Στη συνέχεια προσθέτουμε το στοιχείο.

Η *del_from_pfds* δέχεται τη θέση του file descriptor που θέλουμε να διαγράψουμε και το αντικαθιστά με το τελευταίο στοιχείο του πίνακα.

Όταν ο file descriptor του listening socket δέχεται δεδομένα, αυτό σημαίνει πως υπάρχει μια νέα σύνδεση. Συνεπώς, ο server κάνει *accept* τη νέα σύνδεση και προσθέτει τον file descriptor του νέου socket στον πίνακα απο *pollfd*.

Ταυτόχρονα, όταν ένα απο τα client sockets δέχεται δεδομένα, αυτό σημαίνει πως ο client στέλνει ένα μήνυμα. Συνεπώς, ο server λαμβάνει το μήνυμα, το αποθηκεύει σε έναν προσωπικό buffer και το στέλνει στους υπόλοιπους client.

Για να εγγυηθούμε την ορθή λήψη και αποστολή των μηνυμάτων, υλοποιούμε τις συναρτήσεις *sendall* και *recvall*, οι οποίες στέλνουν και λαμβάνουν αντίστοιχα έναν συγκεκριμένο αριθμό απο χαρακτήρες:

```
void recvall(int fd, char *buf, int *numbytes, int len) {
        int total = 0;
        int bytesleft = len;
        int n;
        while (total < len) {
            n = recv(fd, buf + total, len, 0);
            if (n <= 0) {
                *numbytes = n;
                return;
            }
            total += n;
            bytesleft -= n;
        }
        *numbytes = total;
}

int sendall(int fd, char* buf, int len) {
        int total = 0;
        int bytesleft = len;
        int n;

        while (total < len) {
            n = send(fd, buf + total, bytesleft, 0);
            if (n == -1) {
                return -1;
            }
            total += n;
            bytesleft -= n;
        }
        return 0;
}
```

Ακολουθεί, η λειτουργικότητα που περιγράψαμε παραπάνω:

```
pfds[0].fd = listener;
pfds[0].events = POLLIN; // Report ready on incomming connection
fd_count = 1;

while (1) {
        int poll_count = poll(pfds, fd_count, -1);

        if (poll_count == -1) {
        perror("poll");
        exit(1);
        }
        for (int i = 0; i < fd_count; ++i) {
                if (pfds[i].revents & POLLIN) { // we got one
                if (pfds[i].fd == listener) { // new connection
                addrlen = sizeof remoteaddr;
                newfd = accept(listener, (struct sockaddr*) &remoteaddr, &addrlen);

                if (newfd == -1) {
```

```
                        perror("accept");
                        continue;
                }

                add_to_pfds(&pfds, newfd, &fd_count, &fd_size);
                printf("pollserver: new connection from %s on socket %d\n",
                        inet_ntop(remoteaddr.ss_family, get_in_addr((struct sockaddr*)&remoteaddr),
remoteIP, INET6_ADDRSTRLEN),
                        newfd);
                }
                else { // a client
                int nbytes;
                recvall(pfds[i].fd, buf, &nbytes, MAXDATASIZE);

                if (nbytes <= 0) { // client disconnected
                        if (nbytes == 0) {
                            printf("pollserver: socket %d hung up\n", pfds[i].fd);
                        }
                        else {
                            perror("recv");
                        }
                        close(pfds[i].fd);
                        del_from_pfds(&pfds, i, &fd_count);
                }
                else { // new message
                        for (int j = 0; j < fd_count; ++j) {
                        if (pfds[j].fd != pfds[i].fd && pfds[j].fd != listener) {
                                if (sendall(pfds[j].fd, buf, nbytes) == -1) {
                                perror("send");
                                }
                        }
                        }
                }
                }
                }
        }
}
```

Απο την μεριά του client, χρησιμοποιούμε παρόμοια λογική με τον server. Δηλαδή, έχουμε δυο file descriptors (stdin και server socket) τους οποίους πρέπει να κάνουμε poll ώστε να δούμε αν υπάρχουν νέα δεδομένα.

Όταν έχουμε δεδομένα απο *stdin* τότε τα μορφοποιούμε σε κατάλληλο μέγεθος, τα κρυπτογραφούμε με σχετικό κλειδί (περισσότερα στην επόμενη ενότητα) και τέλος τα στέλνουμε στον server. Η μορφοποίηση σε κατάλληλο μέγεθος γίνεται έτσι ώστε η *recvall* να ξέρει πόσα δεδομένα περιμένει να πάρει. Η άλλη μας επιλογή ήταν να δημιουργήσουμε κάποιο πρωτόκολλο και να πακετάρουμε τα δεδομένα αντίστοιχα, αλλά αυτό ξεφεύγει απο την σκοπιά της παρούσας άσκησης.

Αντίστοιχα, όταν έχουμε δεδομένα απο την μεριά του server, αυτό σημαίνει πως μπορούμε να παραλάβουμε ένα μήνυμα και να το παρουσιάσουμε στον χρήστη. Η λειτουργικότητα φαίνεται παρακάτω:

```
        while (1) {
        int poll_count = poll(pfds, 2, -1);
        if (poll_count == -1) {
        perror("poll");
        exit(1);
        }

        for (int i = 0; i < fd_count; ++i) {
        if (pfds[i].revents & POLLIN) {
                if (pfds[i].fd == STDIN_FILENO) { // client sent a message
                getline(&sent, &size, stdin);
                encrypt(sent, buf, cfd, sid);
                add_pad(sent);

                if (sendall(server, buf, MAXDATASIZE) == -1) {
                        perror("send");
                        exit(1);
                }
                }
                if (pfds[i].fd == server) { // client received a message
                recvall(server, received, &numbytes, MAXDATASIZE);
                if (numbytes <= 0) {
                        if (numbytes == 0) {
                        printf("Server hung up.\n");
                        }
                        else {
                        perror("recv");
                        }
                        exit(1);
                }
                decrypt(received, buf, cfd, sid);
                printf("%s", buf);
                }
        }
        }
        }
```

Παρατηρήστε πως χρησιμοποιούνται οι συναρτήσεις *encrypt* και *decrypt* όταν στέλνουμε και παραλαμβάνουμε ένα μήνυμα αντίστοιχα. Θα τις περιγράψουμε στην επόμενη ενότητα.

2 - Chat Encryption

Σε αυτή την ενότητα θα περιγράψουμε τις συναρτήσεις *encrypt* και *decrypt* που χρησιμοποιήθηκαν απο τον client του chat που υλοποιήσαμε.

Για την κρυπτογράφηση των μηνυμάτων χρησιμοποιούμε το *cryptodev-linux* device. Αυτό το device επιτρέπει σε προγράμματα πρόσβαση στους cryptographic drivers του linux. Συνεπώς, επιτρέπει στα προγράμματα να εκμεταλλευτούν τους επιταχυντές του υλικού οδηγώντας σε καλύτερες επιδόσεις.

Για να χρησιμοποιήσουμε αυτό το device, αρχικά πρέπει να το κάνουμε *open* και στη συνέχεια χρησιμοποιώντας τον file descriptor του να κάνουμε ένα *ioctl* call ώστε να δημιουργήσουμε ένα session. Αυτό υλοποιείται μέσω των ακόλουθων μεθόδων:

```c
unsigned int create_session(int cfd, char *key) {
        struct session_op sess;

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

        sess.cipher = CRYPTO_AES_CBC;
        sess.key = (unsigned char *)key;
        sess.keylen = strlen(key);

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

        return sess.ses;
}

void close_session(int cfd, unsigned int sid) {
        /* Finish crypto session */
        if (ioctl(cfd, CIOCFSESSION, &sid)) {
                perror("ioctl(CIOCFSESSION)");
                exit(1);
        }
}
```

Η *create_session* παίρνει ως παράμετρο τον file descriptor του device και ένα κλειδί το οποίο θα χρησιμοποιηθεί για τις κρυπτογραφικές λειτουργίες σε αυτό το session. Επιστρέφει ένα identification για το session.

Η *close_session* παίρνει ως παράμετρο τον file descriptor του device και το identification του session και εν συνέχεια κλείνει το session.

Έχοντας ένα session μπορούμε λοιπόν να κρυπτογραφήσουμε και να αποκρυπτογραφήσουμε μηνύματα:

```c
void encrypt(char *source, char *dest, int cfd, unsigned int sid) {
        struct crypt_op cryp;
        unsigned char *iv = malloc(sizeof(unsigned char) * BLOCK_SIZE);

        memcpy(iv, "thisisnotrandom", 15);

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

        cryp.ses = sid;
        cryp.len = MAXDATASIZE;
        cryp.src = (unsigned char *)source;
        cryp.dst = (unsigned char *)dest;
        cryp.iv = iv;
        cryp.op = COP_ENCRYPT;

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

void decrypt(char *source, char *dest, int cfd, unsigned int sid) {
        struct crypt_op cryp;
        unsigned char *iv = malloc(sizeof(unsigned char) * BLOCK_SIZE);

        memcpy(iv, "thisisnotrandom", 15);

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

        cryp.ses = sid;
        cryp.len = MAXDATASIZE;
        cryp.src = (unsigned char *)source;
        cryp.dst = (unsigned char *)dest;
        cryp.iv = iv;
        cryp.op = COP_DECRYPT;

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

Οι δυο συναρτήσεις είναι πανομοιότυπες μεταξύ τους, με την έννοια ότι και οι δύο ορίζουν τις ίδιες παραμέτρους στο *struct crypt_op* και χρησιμοποιούν το ίδιο *ioctl* call. Η μόνη διαφορά είναι στο πεδίο *op* του *struct crypt_op* στο οποίο ορίζουμε αμα θέλουμε να κρυπτογραφήσουμε ή να αποκρυπτογραφήσουμε το *src* στο *dst*.

3 - cryptodev Implementation using VirtIO

Σε αυτή την ενότητα θα παρουσιάσουμε την υλοποίηση της εικονικής συσκεύης η οποία μας επιτρέπει να χρησιμοποιήσουμε το cryptodev-linux του host απο ένα guest μηχάνημα.

Η υλοποίηση έχει δύο μέρη:

1. Ένα QEMU crypto backend το οποίο λαμβάνει τα αιτήματα του guest μηχανήματος και πραγματοποιεί κλήσεις προς τον kernel του host.
2. Ένα frontend το οποίο υλοποιείται με οδηγό χαρακτήρων, ο οποίος λαμβάνει τα αιτήματα του userspace του host και τα προωθεί στο QEMU crypto backend μέσω VirtIO.

Θα ξεκινήσουμε με το backend, μιας και η λειτουργία του μοιάζει με τα περιεχόμενα της ενότητας 2.

Το backend λαμβάνει απο τα virtual queues του VirtIO ένα αίτημα το οποίο περιέχει δεδομένα τα οποία είναι read-only και δεδομένα που είναι για write. Εμείς καλούμαστε να διαχειριστούμε τρια είδη αιτημάτων:

1. open
2. close
3. ioctl

Τα *open* και *close* υλοποιούνται σχετικά εύκολα:

```
case VIRTIO_CRYPTODEV_SYSCALL_TYPE_OPEN:
        DEBUG("VIRTIO_CRYPTODEV_SYSCALL_TYPE_OPEN");
        host_fd = elem->in_sg[0].iov_base;
        *host_fd = open("/dev/crypto", O_RDWR);
        break;

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

Ουσιαστικά, όταν δεχόμαστε μια κλήση τύπου open, τότε θέτουμε τον host file descriptor στην τιμή του open file στο backend και επιστρέφουμε. Αντίστοιχα, στο close κλείνουμε τον file descriptor που αρχικοποιήσαμε στο open.

Στις κλήσεις *ioctl* καλούμαστε να διαχειριστούμε τις λειτουργίες σχετικά με την δημιουργία/καταστροφή session και την κρυπτογράφηση/αποκρυπτογράφηση μηνυμάτων:

```
case VIRTIO_CRYPTODEV_SYSCALL_TYPE_IOCTL:
	DEBUG("VIRTIO_CRYPTODEV_SYSCALL_TYPE_IOCTL");

	host_fd = elem->out_sg[1].iov_base;
	ioctl_cmd = elem->out_sg[2].iov_base;
	return_val = elem->in_sg[0].iov_base;
	switch (*ioctl_cmd) {
		case CIOCGSESSION:
			key = elem->out_sg[3].iov_base;
			sop = elem->in_sg[1].iov_base;

			sop->key = key;
			*return_val = ioctl(*host_fd, CIOCGSESSION, sop);
			if (*return_val) {
				perror("ioctl(CIPCCRYPT)");
			}
			break;
		case CIOCFSESSION:
			ses_id = elem->out_sg[3].iov_base;
			*return_val = ioctl(*host_fd, CIOCFSESSION, ses_id);
			if (*return_val) {
				perror("ioctl(CIPCCRYPT)");
			}
			break;
		case CIOCCRYPT:
			cop = elem->out_sg[3].iov_base;
			src = elem->out_sg[4].iov_base;
			iv = elem->out_sg[5].iov_base;

			dst = elem->in_sg[1].iov_base;

			cop->src = src;
			cop->iv = iv;
			cop->dst = dst;

			*return_val = ioctl(*host_fd, CIOCCRYPT, cop);
			if (*return_val) {
				perror("ioctl(CIOCCRYPT)");
			}
			break;
		default:
			*return_val = -1;
	}
	break;
```

Όταν πρόκειται για δημιουργία session, χρειαζόμαστε το κλειδί που θα χρησιμοποιήθει και τα δεδομένα του *struct session_op* τα οποία αποθηκεύουμε στην μεταβλητή *sop*.

Όταν πρόκειται για την καταστροφή ενός session, χρειαζόμαστε μόνο το session id.

Όταν πρόκειται για κρυπτογράφηση/αποκρυπτογράφηση μηνυμάτων, χρησιμοποιούμε το *src* και το *iv* ως input, ενω το *dst* ως output.

Στη συνέχεια κάνουμε τα αντίστοιχα *ioctl* calls και αποθηκεύουμε την τιμή στην μεταβλητή *return_val* η οποία βρίσκεται στο output.

Τέλος, για να ενημερώσουμε το guest process οτι τελειώσαμε την επεξεργασία των δεδομένων που μας έδωσε, χρησιμοποιούμε:

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

Η λειτουργία του frontend περιλαμβάνει την διαχείριση του paravirtualized device driver. Συγκεκριμένα, χρειάζεται να διαχειρίζεται κλήσεις *open*, *close* και *ioctl*.

Η *open* έχει την ευθύνη να διαχειρίζεται το άνοιγμα του αρχείου */dev/cryptodev{0..31}* και να αρχικοποιεί τις διάφορες δομές που θα χρησιμοποιηθούν για τις *ioctl* κλήσεις. Επίσης, πρέπει να προωθεί την κλήση *open* στο backend έτσι ώστε να γίνει *open* το αρχείο */dev/crypto* του host μηχανήματος και να επιστραφεί ο file descriptor που του αντιστοιχίζεται. Η υλοποίηση της *open* είναι:

```
static int crypto_chrdev_open(struct inode *inode, struct file *filp)
{
        int ret = 0;
        int err;
        unsigned int len;
        struct crypto_open_file *crof;
        struct crypto_device *crdev;
        struct virtqueue *vq;
        unsigned int *syscall_type;
        unsigned int num_out, num_in;
        int *host_fd;
        struct scatterlist syscall_type_sg, host_fd_sg, *sgs[2];

        num_out = num_in = 0;

        debug("Entering");

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

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

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

        /* Associate this open file with the relevant crypto device. */
        crdev = get_crypto_dev_by_minor(iminor(inode));
        if (!crdev) {
                debug("Could not find crypto device with %u minor",
                      iminor(inode));
                ret = -ENODEV;
                goto fail;
        }

        crof = kzalloc(sizeof(*crof), GFP_KERNEL);
        if (!crof) {
                ret = -ENOMEM;
                goto fail;
        }
```

```
        crof->crdev = crdev;
        crof->host_fd = -1;
        filp->private_data = crof;

        vq = crdev->vq;
        sema_init(&crdev->lock, 1);

        /**
        * We need two sg lists, one for syscall_type and one to get the
        * file descriptor from the host.
        **/
        sg_init_one(&syscall_type_sg, syscall_type, sizeof(*syscall_type));
        sgs[num_out++] = &syscall_type_sg;
        sg_init_one(&host_fd_sg, host_fd, sizeof(*host_fd));
        sgs[num_out + num_in++] = &host_fd_sg;

        /**
        * Wait for the host to process our data.
        **/
        if (down_interruptible(&crdev->lock)) {
                return -ERESTARTSYS;
        }
        err = virtqueue_add_sgs(vq, sgs, num_out, num_in,
                                &syscall_type_sg, GFP_ATOMIC);
        virtqueue_kick(vq);

        while (virtqueue_get_buf(vq, &len) == NULL) {
                /* do nothing */;
        }
        up(&crdev->lock);

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

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

Αφού βρούμε το συγκεκριμένο device στο οποίο αναφέρεται η κλήση open, αναθέτουμε μνήμη στις διάφορες δομές που θα χρησιμοποιηθούν. Συγκεκριμένα, η δομή *struct crypto_open_file* κρατάει πληροφορίες σχετικά με το αρχείο που άνοιξε και η δομή *struct crypto_device* κρατάει πληροφορίες σχετικά με το device που χρησιμοποιήθηκε.

Στη συνέχεια, αρχικοποιούμε τα δεδομένα που θα στείλουμε στο backend ώστε να ανοίξει το */dev/crypto* και τα δεδομένα που θέλουμε να μας επιστραφούν, δηλαδή τον file descriptor που πήρε ο host. Αυτή η λειτουργία επιτυγχάνεται με την χρήση των virtual queues του VirtIO και μέσω της δομής *struct scatterlist*. Συγκεκριμένα, δίνουμε τους pointers για τις θέσεις μνήμης των δεδομένων που θέλουμε να ανταλλάξουμε και το VirtIO διαχειρίζεται την μεταφορά τους.

Αφού μεταφέρουμε τα δεδομένα (*virtqueue_add_sgs*), ενημερώνουμε το backend μέσω της κλήσης *virtqueue_kick* και περιμένουμε να τα επεξεργαστεί μέσω της κλήσης *virtqueue_get_buf*. Τέλος, ελέγχουμε αμα το backend μας επέστρεψε ένα valid file descriptor και τον αποθηκεύουμε.

Σε όλες τις περιπτώσεις (ioctl, open, release), κάνουμε lock ένα mutex προτού στείλουμε και λάβουμε τα δεδομένα, έτσι ώστε να αποφύγουμε race conditions στο virtual queue.

Όταν θέλουμε να κλείσουμε το αρχείο, ακολουθούμε την ίδια διαδικασία για να στείλουμε στο backend τον file descriptor που αποθηκεύσαμε, ώστε να κλείσει το */dev/crypto*:

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

        num_out = num_in = 0;

        debug("Entering");

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

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

        /**
        * Wait for the host to process our data.
        **/
        if (down_interruptible(&crdev->lock)) {
                return -ERESTARTSYS;
        }
        err = virtqueue_add_sgs(vq, sgs, num_out, num_in,
                                &syscall_type_sg, GFP_ATOMIC);
        virtqueue_kick(vq);
        while (virtqueue_get_buf(vq, &len) == NULL)
                /* do nothing */;
        up(&crdev->lock);

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

}
```

Για τις *ioctl* κλήσεις, όπως και στο backend πρέπει να διακρίνουμε τις διάφορες περιπτώσεις:

1. Άνοιγμα session
2. Κλείσιμο session
3. Κρυπτογράφηση/Αποκρυπτογράφηση μηνύματος

Ταυτόχρονα, μιας και οι ioctl κλήσεις παίρνουν απροσδιόριστα δεδομένα απο τον χρήστη, χρειαζόμαστε ασφαλή μηχανισμό μεταφοράς δεδομένων μεταξύ user space και kernel space. Σε αυτό θα μας φανούν χρήσιμες οι *copy_{from,to}_user*.

Αφού αρχικοποιήσουμε τις διάφορες δομές (παραλείπονται για λόγους χώρου), βάζουμε στα *scatterlist* τα δεδομένα που πρέπει να σταλούν σε κάθε *ioctl* call:

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

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

        sg_init_one(&ioctl_cmd_sg, send_cmd, sizeof(*send_cmd));
        sgs[num_out++] = &ioctl_cmd_sg;
```

Συγκεκριμένα, στέλνουμε πάντα τον file descriptor στον οποίο αναφερόμαστε απο μεριάς host και το *ioctl* command που θέλουμε να εκτελέσει το backend.

Στη συνέχεια, εισάγουμε τα δεδομένα που είναι συγκεκριμένα για κάθε *ioctl* call.

Αρχικά, όταν θέλουμε να δημιουργήσουμε ένα session, ο χρήστης μας δίνει μια δομή *struct session_op* ως παράμετρο που πρέπει να αντιγράψουμε. Επίσης, μιας και η δομή *struct session_op* περιέχει έναν pointer στο κλειδί που επιθυμεί ο χρήστης να χρησιμοποιήσει, πρέπει να το αντιγράψουμε και αυτό μιας και τα δεδομένα που δείχνει ο pointer βρίσκονται σε user space:

```
        case CIOCGSESSION:
                debug("CIOCGSESSION");
                if (copy_from_user(sop, (struct session_op __user *) arg, sizeof(struct
session_op))) {
                        ret = -EINVAL;
                        goto fail;
                }
                if (copy_from_user(key, sop->key, sop->keylen)) {
                        ret = -EINVAL;
                        goto fail;
                }
                sg_init_one(&key_sg, key, sizeof(*key));
                sgs[num_out++] = &key_sg;

                sg_init_one(&input_msg_sg, retval, sizeof(*retval));
                sgs[num_out + num_in++] = &input_msg_sg;

                sg_init_one(&output_msg_sg, sop, sizeof(struct session_op));
                sgs[num_out + num_in++] = &output_msg_sg;

                break;
```

Θέλουμε να γίνει modify το *struct session_op* που έχουμε και η τιμή που μας επιστρέφει το backend, συνεπώς τα βάζουμε στα input buffers. Ταυτόχρονα, το κλειδί θέλουμε να γίνει μόνο read, και συνεπώς το βάζουμε στα output buffers.

Όταν μας επιστραφούν δεδομένα απο το backend, θέλουμε να μεταφέρουμε το modified *struct session_op* πίσω στον χρήστη, μιας και η κλήση *ioctl* θέτει κάποια πεδία του. Για να το καταφέρουμε αυτό, χρησιμοποιούμε την *copy_to_user*:

```
        case CIOCGSESSION:
                if (copy_to_user((struct session_op __user *) arg, sop, sizeof(struct session_op)))
{
                        ret = -EINVAL;
                        goto fail;
                }
                break;
```

Όταν θέλουμε να κλείσουμε το session, η παράμετρος του *ioctl* call είναι το session id. Συνεπώς, το αντιγράφουμε απο τον χώρο χρήστη, και το προωθούμε στο backend:

```
        case CIOCFSESSION:
                debug("CIOCFSESSION");
                if (copy_from_user(&sessid, (void __user *) arg, sizeof(int))) {
                        ret = -EINVAL;
                        goto fail;
                }
                sg_init_one(&output_msg_sg, &sessid, sizeof(int));
                sgs[num_out++] = &output_msg_sg;

                sg_init_one(&input_msg_sg, retval, sizeof(*retval));
                sgs[num_out + num_in++] = &input_msg_sg;
                break;
```

Δεν χρειάζεται να επιστρέψουμε τίποτα στον χώρο χρήστη και συνεπώς όταν το backend επεξεργαστεί τα δεδομένα, επιστρέφουμε.

Τέλος, όταν θέλουμε να κρυπτογραφήσουμε/αποκρυπτογραφήσουμε ένα μήνυμα η παράμετρος που δίνεται στην κλήση *ioctl* είναι ένας pointer στην δομή *struct crypt_op*. Αντιγράφουμε αυτή την δομή στο χώρο πυρήνα και στην συνέχεια αντιγράφουμε τα πεδία *src, dst, iv* που περιέχει. Στη συνέχεια, τα προωθούμε στο backend:

```
        case CIOCCRYPT:
                debug("CIOCCRYPT");
                if (copy_from_user(cop, (void __user *) arg, sizeof(struct crypt_op))) {
                        ret = -EINVAL;
                        goto fail;
                }
                saved_cop_dst = cop->dst;
                if (cop->iv) {
                        if (copy_from_user(iv, cop->iv, IV_SIZE)) {
                                ret = -EINVAL;
                                goto fail;
                        }
                }
                if (copy_from_user(src, cop->src, DATA_SIZE)) {
                        ret = -EINVAL;
                        goto fail;
                }
                if (copy_from_user(dst, cop->dst, DATA_SIZE)) {
                        ret = -EINVAL;
                        goto fail;
                }

                sg_init_one(&output_msg_sg, cop, sizeof(struct crypt_op));
```

```
        sgs[num_out++] = &output_msg_sg;

        sg_init_one(&src_sg, src, sizeof(*src));
        sgs[num_out++] = &src_sg;

        sg_init_one(&key_sg, iv, sizeof(*iv));
        sgs[num_out++] = &key_sg;

        sg_init_one(&input_msg_sg, retval, sizeof(*retval));
        sgs[num_out + num_in++] = &input_msg_sg;

        sg_init_one(&dst_sg, dst, sizeof(*dst));
        sgs[num_out + num_in++] = &dst_sg;
        break;
```

Θέλουμε το *dst* να επιστραφεί στον χρήστη, και συνεπώς αφού επιστραφούν δεδομένα απο το backend έχουμε την ακόλουθη κλήση:

```
    case CIOCCRYPT:
        if ((res = copy_to_user((unsigned char __user *)saved_cop_dst, dst, DATA_SIZE))) {
            ret = -EINVAL;
            goto fail;
        }
        break;
```

Για να στείλουμε/λάβουμε δεδομένα απο/προς το backend χρησιμοποιούμε την ίδια μεθοδολογία όπως και με *open/close*.