Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Εργαστήριο Λειτουργικών Συστημάτων
Ακαδημαϊκό έτος 2012-2013

Αναφορά Εργαστηριακών Ασκήσεων

Μιχάλης Ροζής 03109704
Νίκος Τερλεμές 03109110

1η εργαστηριακή άσκηση
Οδηγός ασύρματου δικτύου αισθητήρων

Στην πρώτη εργαστηριακή άσκηση μας ζητείται η υλοποίηση οδηγού χαρακτήρων για την απεικόνιση των μετρήσεων ενός ασύρματου δικτύου αισθητήρων. Το ασύρματο δίκτυο αποτελείται από αισθητήρες που ο καθένας καταγράφει τρεις μετρήσεις, τάσης, φωτεινότητας και θερμοκρασίας. Σκοπός της άσκησης είναι η υλοποίηση οδηγού πάνω στο στρώμα TTY ώστε να γίνεται δυνατή η απεικόνιση στο χώρο χρήστη της κάθε μέτρησης όλων των αισθητήρων ανεξάρτητα απο τις υπόλοιπες ως συσκευή χαρακτήρων στον κατάλογο /dev.

Περιγραφή υλοποίησης

Για την υλοποίηση του οδηγού είναι αναγκαία η τροποποίηση του υπάρχοντος στρώματος TTY ώστε η συλλογή και επεξεργασία των δεδομένων να μην ακολουθεί τη συνηθισμένη πορεία του πρωτοκόλλου αλλά να επεξεργάζεται απο το δικό μας line discipline και το στρώμα συλλογής και επεξεργασίας δεδομένων. Κάθε μέτρηση κάθε αισθητήρα θα πρέπει να απεικονίζεται σε μια ξεχωριστή συσκευή χαρακτήρων οπότε κλήσεις open, read σε αυτή τη συσκευή θα πρέπει να απεικονίζουν τις αντίστοιχες μετρήσεις σε δεκαδική τιμή στο χώρο χρήστη.

Αρχικά θα πρέπει να δημιουργηθούν και να καταχωρηθούν οι συσκευές χαρακτήρων στο σύστημα με το κατάλληλο major και minor number. Αυτό επιτυγχάνεται με τη συνάρτηση lunix_chrdev_init η οποία καλείται κατά την εισαγωγή του module στον πυρήνα του linux. Αρχικά επιτυγχάνεται με τον ακόλουθο κώδικα:

```
    int ret;
dev_t dev_no;
unsigned int lunix_minor_cnt = lunix_sensor_cnt << 3;

debug("initializing character device\n");
cdev_init(&lunix_chrdev_cdev, &lunix_chrdev_fops);
lunix_chrdev_cdev.owner = THIS_MODULE;

dev_no = MKDEV(LUNIX_CHRDEV_MAJOR, 0);
ret = register_chrdev_region(dev_no,lunix_sensor_cnt,"lunix");
if (ret < 0) {
      debug("failed to register region, ret = %d\n", ret);
      goto out;
}
```

```
	ret = cdev_add(&lunix_chrdev_cdev,dev_no,lunix_sensor_cnt);
	if (ret < 0) {
		debug("failed to add character device\n");
		goto out_with_chrdev_region;
	}
	debug("completed successfully\n");
	return 0;

out_with_chrdev_region:
	unregister_chrdev_region(dev_no, lunix_minor_cnt);
out:
	return ret;
```

Η cdev_init αρχικοποιεί τη συσκευή χαρακτήρων με τα αντίστοιχα file operations που έχουμε ορίστει στη δομή file_operations. Στη συνέχεια καταχωρούμε ένα εύρος lunix_sensor_cnt (16) συσκευών χαρακτήρων με major number το 60 και αρχική τιμή minor το 0 και στη συνέχεια ενεργοποιούμε τις συσκευές χαρακτήρων με την εντολή cdev_add.

Στη συνέχεια εξετάζουμε τη λειτουργία open του οδηγού συσκευών. Όταν πραγματοποιούμε open στη συσκευή /dev/lunix* εκτελείται η συναρτηση lunix_chrdev_open εξαιτίας των συσχετισμένων file_operations απο την lunix_chrdev_init. Η open δέχεται ως παράμετρο τη δομή file και inode του συγκεκριμένου κόμβου στο /dev. Κάθε συσκευή lunix αναφέρεται σε μια απο τις 3 μετρήσεις οπότε κατα τη λειτουργία open πρέπει να εξετάσουμε σε ποια μέτρηση αντιστοιχίζεται το συγκεκριμένο inode. Αυτό επιτυγχάνεται απο τον έλεγχο του minor number σύμφωνα με τη σύμβαση που έχουμε υιοθετήσει. Έτσι το υπόλοιπο της διαίρεσης του minor number με το 8 μας δείχνει το είδος της μέτρησης (0 batt,1 temp, 2 light). Κατα την κλήση της open φροντίζουμε να δημιουργήσουμε τη δομή chrdev_state_struct η οποία περιέχει πληροφορίες σχετικά με την τρέχουσα κατάσταση της συσκευής χαρακτήρων, να ενημερώσουμε τη δομή για το είδος της μέτρησης και τον αισθητήρα και να αρχικοποιήσουμε το σημαφόρο της συσκευής. Τέλος φροντίζουμε να θέσουμε το πεδίο private_data της δομής file ως δείκτη στη δομή αυτή. Τα παραπάνω φαίνονται στο ακόλουθο κομμάτι κώδικα για την open:

```
	 struct lunix_chrdev_state_struct *chrdev;
	chrdev = (lunix_chrdev_state_struct *) kmalloc(sizeof(struct
lunix_chrdev_state_struct),GFP_KERNEL);
	unsigned int msr;
	int ret;

	debug("entering\n");
	ret = -ENODEV;
	if ((ret = nonseekable_open(inode, filp)) < 0)
		goto out;

	msr = iminor(inode) % 8;
	chrdev->type = msr;
	chrdev->sensor = lunix_sensors[si];
	init_MUTEX(&chrdev->lock);
	filp->private_data = chrdev;
```

```
out:
	debug("leaving, with ret = %d\n", ret);
	return ret;
```

Στη συνέχεια αναλύεται η λειτουργία της κλήσης read. Όταν πραγματοποιείται κλήση συστήματος read στη συσκευή /dev/lunix* τότε καλείται η lunix_chrdev_read. Αρχικά παίρνουμε το σημαφόρο ώστε μόνο μια διεργασία να μπορεί να πραγματοποιήσει κάθε φορά λειτουργία read πάνω στη συσκευή χαρακτήρων. Στη συνέχεια ελέγχουμε τη θέση ανάγνωσης της δομής file ώστε να ξέρουμε αν έχει ολοκληρωθεί ή όχι κάποια προηγούμενη λειτουργία read. Αυτό γίνεται ελέγχοντας τη μεταβλητή f_pos. Αν είναι 0 τότε γνωρίζουμε οτι η θέση ανάγνωσης είναι στην αρχή οπότε η προηγούμενη λειτουργία έχει ολοκληρωθεί. Στη συνέχεια ελέγχουμε αν έχουμε δεχτεί καινούργια δεδομένα απο το προηγούμενο στρώμα.

Αυτό γίνεται χρησιμοποιώντας τη βοηθητική συνάρτηση lunix_chrdev_state_update. Αρχικά ελέγχουμε αν συμφωνούν οι τιμές buf_timestamp της τρέχουσας κατάστασης της συσκευής και last_update του αισθητήρα. Αν είναι ίσες τότε ξέρουμε οτι δεν υπάρχουν αλλαγές οπότε συνάρτηση επιστρέφη EGAIN και η διεργασία κοιμάται μέχρι να φτάσουν καινούργιες τιμές, δηλαδή ανανεωθούν τα timestamps. Επειδή η ενημέρωση της μεταβλητής last_update καθώς και των μετρήσεων γίνεται απο το υλικό ασύγχρονα, δηλαδή σε interrupt context, πρέπει να φροντίσουμε για την απόκτηση του spinlock του αντίστοιχου αισθητήρα πριν τον έλεγχο του last_update . Σε αντίθετη περίπτωση, υπάρχει η πιθανότητα το κάτω στρώμα να προσπαθήσει να τροποποιήσει το κρίσιμο τμήμα κατα τη διάρκεια της ανάγνωσής του απο την κλήση read οπότε η τιμές που τελικά θα διαβαστούν να είναι “σκουπίδια”.

Μόλις ανανεωθούν οι τιμές των αισθητήρων απο το κάτω στρώμα, τότε προκαλείται μια διακοπή υλικού και η διεγασία που είχε κοιμηθεί περιμένοντας για αλλαγές ξυπνάει. Στη συνέχεια αποκτούμε τη νέα τιμή της μέτρησης, ενημερώνουμε τα timestamps και ελευθερώνουμε το spinlock.

Πλεον μπορούμε να μορφοποιήσουμε τη μέτρηση με τον κατάλληλο τρόπο απλά κρατώντας το σημαφόρο που είχαμε αποκτήσει. Η μορφοποίηση γίνεται χρησιμοποιώντας lookup πίνακες και κρατώντας κατάλληλα το ακέραιο και δεκαδικό μέρος. Κάθε ψηφίο της μέτρησης αποθηκεύεται στον πίνακα buf_data και χρησιμοποιούμε το buf_lim για την σωστή μεταφορά των ζητούμενων δεδομένων στο χώρο χρήστη. Ο κώδικας της lunix_chrdev_state_update δίδεται παραπάνω.

```
static int lunix_chrdev_state_update(struct lunix_chrdev_state_struct *state)
{
	struct lunix_sensor_struct *sensor;
	uint16_t value;
	long unformatted;
	int i,order,x;
	i=4;
	order=1;
	sensor=state->sensor;
	spin_lock_irq(&sensor->lock);


	if (state->buf_timestamp == sensor->msr_data[state->type]->last_update)
```

```
            return -EAGAIN;

        value=sensor->msr_data[state->type];
        state->buf_timestamp = sensor->msr_data[state->type]->last_update;
        spin_unlock_irq(&sensor->lock);

        unformatted=lookup_temperature[value];
        if (unformatted <0)
            state->buf_data[0]='-';
        else state->buf_data[0]='+';
        x=unformatted;
        while (x>1) {
            x=x/10;
            order*=10;
        }
        buf_data[1]=unformatted/order + '0';
        x=unformatted % order;
        order /= 10;
        buf_data[2]=x/order + '0';
        buf_data[3]='.';
        while (order>1) {
            x = x % order;
            order/=10;
            buf_data[i]=x/order + '0';
            i++
        }
        buf_lim = i;



        debug("leaving\n");
        return 0;
}
```

Πλεον ο πίνακας buf_data περιέρχει τα απαραίτητα δεδομένα τα οποία θα αποστείλουμε στο χώρο χρήστη με την copy_to_user και το μόνο που απομένει είναι να καθορίσουμε τα ακριβή όρια των δεδομένων που θα στείλουμε. Αυτό γίνεται συγκρίνοντας τα δεδομένα που ζητήθηκαν απο το χρήστη (μεταβλητή cnt) και τα δεδομένα που έχουμε (buf_lim). Στην περίπτωση που η προηγούμενη λειτουργία read έχει ολοκληρωθεί, δηλαδή έχουμε αποστείλει όλα τα δεδομένα μιας μέτρησης στο χώρο χρήστη, μεταφέρουμε τα δεδομένα που ζητήθηκαν. Αν δεν έχουμε τον ζητούμενο όγκο δεδομένων, μεταφέρουμε όσα έχουμε διαθέσιμα. Αν ζητούνται απο το χώρο χρήστη λιγότερα δεδομένα απο τα δεδομένα που έχουμε διαθέσιμα, τότε τα στέλνουμε αλλά φροντίζουμε για την αύξηση του f_pos ώστε η επόμενη λειτουργία read να συνεχίσει απο το σημείο που είχε μείνει η προηγούμενη.

Σε περίπτωση που η προηγούμενη λειτουργία read δεν έχει ολοκληρωθεί, τότε αποστέλνουμε όσα δεδομένα έχουμε διαθέσιμα αλλά ξεκινώντας απο το σημείο buf_data + f_pos δηλαδή απο το σημείο που είχαμε μείνει στην προηγούμενη κλήση της read και μεταφέρουμε αντίστοιχα είτε όσα έχουμε διαθέσιμα είτε όσα ζητούνται.

Τέλος φροντίζουμε για την απελευθέρωση του σημαφόρου.

Η περιγραφή των παραπάνω οδηγεί στον ακόλουθο κώδικα για τη read

```
static ssize_t lunix_chrdev_read(struct file *filp, char __user *usrbuf, size_t cnt, loff_t *f_pos)
{
    ssize_t ret;
```

```
	struct lunix_sensor_struct *sensor;
	struct lunix_chrdev_state_struct *state;

	state = filp->private_data;
	WARN_ON(!state);

	sensor = state->sensor;
	WARN_ON(!sensor);

	if (down_interruptible(&state->lock));
	if (*f_pos == 0) {
		while (lunix_chrdev_state_update(state) == -EAGAIN) {

			up(&state->lock);
			if (wait_event_interruptible(sensor->wq,state->buf_timestamp!=sensor->msr_data[state-
>type]->last_update))
				return -ERESTARTSYS;
			if (down_interruptible(&state->lock));
		}
		if (state->buf_lim > cnt) {
			copy_to_user(usrbuf,state->buf_data,cnt);
			   ret=cnt
			*f_pos+=cnt;
			goto out;
		}
		else {
			copy_to_user(usrbuf,state->buf_data,buf_lim);
			   ret = buf_lim
			goto out;
		}
	}
	if ((buf_lim - *f_pos)> cnt) {
		copy_to_user(usrbuf,state->(buf_data+(*f_pos)),cnt);
		  ret = cnt;
		*f_pos += cnt;
	}
	else {
		copy_to_user(usrbuf,state->(buf_data+(*f_pos)),(buf_lim - *f_pos));
		  ret = buf_lim - *f_pos;
		*f_pos = 0;
	}

out:
	up(&state->lock);
	return ret;
}
```

Τέλος τροποποιούμε την lunix_chrdev_release ώστε να απελευθερώνει τους πόρους που κάναμε allocate κατα την κλήση της open.

2η εργαστηριακή άσκηση

Chat session μεταξύ δυο peers

Επέκταση για υποστήριξη κρυπτογραφημένων μηνυμάτων

Κρυπτογραφική Συσκευή VirtIO για QEMU-KVM

Η 2η εργαστηριακή άσκηση περιλαμβάνει τρία ζητούμενα:

- Στο πρώτο ζητούμενο μας ζητείται να υλοποιήσουμε μια εφαρμογή chat για αμφίδρομη επικοινωνία μεταξύ δυο peers. Η επικοινωνία θα επιτυγχάνεται με χρήση του BSD Socket API και τα δεδομένα δεν θα υφίστανται καμία κρυπτογράφηση.
- Στο δεύτερο μέρος της άσκησης μας ζητείται να επεκτείνουμε το πρώτο ζητούμενο ώστε τα δεδομένα να κινούνται πάνω στο δίκτυο κρυπτογραφημένα. Η κρυπτογράφηση και η αποκρυπτογράφησή τους θα πραγματοποιείται από την εικονική συσκευή cryptodev που βρίσκεται στον αποστολέα καθως και στον παραλήπτη.
- Στο τρίτο και τελευταίο ζητούμενο της άσκησης, επεκτείνουμε τα προηγούμενα ζητήματα ώστε η κρυπτογράφηση να πραγματοποιείται στο "εξωτερικό μηχάνημα". Συγκεκριμένα, χρησιμοποιούμε virtualization μέσω του QEMU-KVM ώστε να δημιουργήσουμε ένα εικονικό μηχάνημα (guest) μέσα στο ήδη υπάρχον μηχάνημα (host). Στο guest μηχάνημα πραγματοποιούνται κλήσεις ioctl προς μια εικονική κρυπτογραφική συσκευή οι οποίες αντιστοιχίζονται σε κλήσεις ioctl στη συσκευή cryptodev που χρησιμοποιήθηκε στο 2ο ζήτημα και βρίσκεται στο host μηχάνημα.

Ζητούμενο 1ο):

Στο πρώτο ζητούμενο της άσκησης υλοποιούμε μια εφαρμογή chat η οποία υποστηρίζει επικοινωνία peer-to-peer μεταξύ μόνο δυο peers και δεν χρησιμοποιείται μεσολαβητής – server για την εγκαθίδρυση της σύνδεσης. Στην υλοποίησή μας, και τα δυο peers έχουν ίσες ευκαιρίες ως προς το hosting του session. Με τον όρο hosting εννοούμε ότι ένα μέλος της συνεδρίας θα λειτουργεί ως server, δηλαδή θα έχει ρυθμίσει κατάλληλα ένα socket και θα "ακούει" σε αυτό για νέες αιτήσεις σύνδεσης.

Περιγραφή υλοποίησης

Η βάση της υλοποίησής μας είναι η αμοιβαία δυνατότητα για hosting του chat session με επαναλαμβανόμενες δοκιμές ως client και ως server. Με τον όρο client εννοούμε την πραγματοποίηση κλήσεων συστήματος connect (αίτηση για σύνδεση σε συγκεκριμένη διεύθυνση και θύρα) και με τον όρο server εννοούμε την πραγματοποίηση κλήσεων bind, listen και accept, δηλαδή την αναμονή για αιτήσεις συνδέσεων απο άλλες διεργασίες. Αρχικά, το πρώτο μέλος της συνομιλίας που θέλει να επικοινωνήσει με το δεύτερο μέλος, δοκιμάζει να συνδεθεί σε μια προκαθορισμένη θύρα (στην περίπτωσή μας 35001) και στη διεύθυνση του άλλου μέλους. Δηλαδή κάνει μια κλήση της connect πάνω σε ένα socket για TCP/IP πρωτόκολλο. Αν η κλήση αποτύχει, αυτό σημαίνει οτι δεν υπάρχει κανείς ο οποίος ακούει για συνδέσεις στο άλλο άκρο. Τότε δοκιμάζει να γίνει host για το συγκεκριμένο session, δηλαδή ετοιμάζει ένα server configuration μέσω κλήσεων στις bind, listen και accept. Πλέον, δηλαδή, ακούει σε μια συγκεκριμένη θύρα για εισερχόμενες συνδέσεις. Ύστερα από ορισμένο διάστημα (στην υλοποίησή μας έχει

οριστεί στα 3 δευτερόλεπτα) κατά το οποίο δεν έχει πραγματοποιηθεί αίτηση για σύνδεση από άλλη εφαρμογή, το πρώτο μέλος ξαναδοκιμάζει να συνδεθεί ως client πλέον στην ίδια διεύθυνση και θύρα της πρώτης προσπάθειας. Σε περίπτωση που πραγματοποιηθεί αίτηση για σύνδεση τότε η εγκατάσταση σύνδεσης γίνεται επιτυχώς. Αν πάλι αποτύχει η σύνδεση δοκιμάζει πάλι ως server και η διαδικασία αυτή επαναλαμβάνεται συνεχώς. Παράλληλα, τον ίδιο κώδικα τρέχει και το δεύτερο μέλος της συνεδρίας, δηλαδή τη μια προσπαθεί να συνδεθεί στο άλλο μηχάνημα (ή εφαρμογή) και την άλλη προσπαθεί να ακούσει για αιτήσεις για σύνδεση από άλλα μηχανήματα ή εφαρμογές.

Θεωρητικά, για να αποτύχει η εγκαθίδρυση σύνδεσης μεταξύ των δυο μελών της συνεδρίας θα πρέπει να δοκιμάζουν ταυτόχρονα και οι δυο διαδικασίες να συνδεθούν ως client και ταυτόχρονα ως server, δηλαδή οι κλήσεις της connect και της accept να γίνονται ταυτόχρονα επ' άπειρον. Συνυπολογίζοντας τις διαφορές στην αρχιτεκτονική των μηχανημάτων, στη συχνότητα των ρολογιών, στο scheduling που υφίστανται από τον χρονοδρομολογητή, φαίνεται μάλλον απίθανο να αποτύχει η εγκατάσταση σύνδεσης μεταξύ των δυο εφαρμογών.

Μια εναλλακτική υλοποίηση του chat είναι το κάθε μέλος να δοκιμάζει πρώτα να κάνει σύνδεση ως client (με κλήση στην connect) και αν αποτύχει να δοκιμάζει να γίνει host της συνεδρίας (περιμένοντας αίτηση για σύνδεση) και να παραμένει σε server configuration για πάντα. Αυτή η προσέγγιση είναι πιο εύκολη στην υλοποίησή της αλλά η πιθανότητα να αποτύχει η εγκατάσταση σύνδεσης είναι αυξημένη καθώς αρκούν δυο ταυτόχρονες δοκιμές ως client από το κάθε μέλος ώστε να μπουν και τα δυο μέλη σε server configuration, αναμένοντας για νέες αιτήσεις σύνδεσης.

Λεπτομέρειες υλοποίησης

Όπως αναφέρθηκε παραπάνω, η κλήση accept μπλοκάρει αναμένοντας για καινούργιες αιτήσεις, κάτι το οποίο μειώνει την λειτουργικότητα της υλοποίησής μας. Για το λόγο αυτό χρησιμοποιούμε την κλήση συστήματος select του UNIX η οποία παρακολουθεί ένα σύνολο από file descriptors και με τη χρήση κατάλληλων μακροεντολών ενημερώνει για το πότε ένας file descriptor είναι έτοιμος για μια συγκεκριμένη κλήση συστήματος πάνω σε αυτόν. Έτσι αντί να καλέσουμε κατευθείαν την accept και να μπλοκάρει μέχρι να έρθει αίτηση για σύνδεση, καλούμε την select και αυτή μας ενημερώνει πότε ο file descriptor που αντιστοιχεί στο socket έχει δεχτεί αίτημα για σύνδεση. Χωρίς τη χρήση της select η υλοποίησή μας θα αποτύγχανε να εγκαταστήσει σύνδεση μεταξύ των δυο συνομιλητών διότι αυτή βασίζεται στην επαναλαμβανόμενη προσπάθεια σύνδεσης ως client και ως server.

Αρχικά παρατίθενται για πληρότητα οι δηλώσεις μεταβλητών και κάποιες αρχικοποιήσεις:

```
int sd,port,newsd,flag;
ssize_t n;
```

```
    socklen_t len;
    char buf[100];
    char *hostname;
    struct hostent *hp;
    struct sockaddr_in sa,sa_server;
    fd_set rfds;
    struct timeval tv;
    char addrstr[INET_ADDRSTRLEN];

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

Πλέον η μεταβλητές port και hostname περιλαμβάνουν τη θύρα και τη διεύθυνση στην οποία επιχειρούμε να συνδεθούμε.

Στη συνέχεια δημιουργούμε ένα socket για TCP/IP στη συγκεκριμένη διεύθυνση και θύρα μέσω της συνάρτησης client_init.

```
int client_init(struct sockaddr_in *sa_client, int port, struct hostent *hp) {
    int sd;
    if ((sd = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
        perror("socket");
        exit(1);
    }
    fprintf(stderr, "Created TCP socket\n");
    sa_client->sin_family = AF_INET;
    sa_client->sin_port = htons(port);
    memcpy(&(sa_client->sin_addr.s_addr), hp->h_addr, sizeof(struct in_addr));
    return sd;
}
```

Η συνάρτηση αυτή δημιουργεί ένα καινούργιο socket το οποίο έχει ρυθμιστεί για TCP/IP πρωτόκολλο και αρχικοποιεί τη δομή sockaddr_in με πεδία family=INET,sin_port=htons(port) και sin_addr τη διεύθυνση που εισάγαμε ως argument στην εφαρμογή, την οποία εχουμε επεξεργαστεί προηγουμένως κατάλληλα. Η δομή αυτή θα δοθεί ως παράμετρος στη συνάρτηση connect η οποία θα προσπαθήσει να εγκαθιδρύσει σύνδεση στη συγκεκριμένη διεύθυνση και πόρτα. Η συνάρτηση επιστρέφει τον file descriptor που αντιστοιχεί στο συγκεκριμένο socket.

Ακολούθως δίνεται η συνάρτηση server_init η οποία έχει την ευθύνη για την κατάλληλη παραμετροποίηση ώστε η διαδικασία να λειτουργεί ως server της συνεδρίας.

```
int server_init(struct sockaddr_in *sa_server) {
```

```
    int sd;
    if ((sd = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
        perror("socket");
        exit(1);
    }
    memset(sa_server, 0, sizeof(*sa_server));
    sa_server->sin_family = AF_INET;
    sa_server->sin_port = htons(TCP_PORT);
    sa_server->sin_addr.s_addr = htonl(INADDR_ANY);
    if (bind(sd, (struct sockaddr *)sa_server, sizeof(*sa_server)) < 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);
    }
    return sd;
}
```

Αρχικά η συνάρτηση δημιουργεί socket πάνω απο TCP και αρχικοποιεί τη δομή sockaddr_in ώστε να αναφέρεται σε διευθύνσεις internet, στην καθορισμένη πόρτα TCP_PORT=35001 και στη διεύθυνση INADDR_ANY η οποία αντιστοιχεί στη διεύθυνση 0.0.0.0. Αυτό σημαίνει οτι η διαδικασία θα ακούει για αιτήσεις απο οποιαδήποτε δυνατή διεύθυνση. Στη συνέχεια εκτελείται η κλήση bind η οποία αναθέτει τη συγκεκριμένη διεύθυνση και θύρα στο socket μέσω του file descriptor. Τέλος καλείται η listen με παράμετρο TCP_BACKLOG=5, το οποίο αντιστοιχεί στο μέγιστο μέγεθος ενεργών συνδέσεων. Η server_init επιστρέφει τον file descriptor στον οποίο θα εκτελεί έλεγχο η accept για πιθανές νέες συνδέσεις.

Το βασικό τμήμα της εγκατάστασης της σύνδεσης εμπεριέχεται στον ακόλουθο ατέρμον βρόγχο, στον οποίο εναλλάσονται οι προσπάθειες για σύνδεση ως client και ως server:

```
while (1) {
        if (connect(sd, (struct sockaddr *) &sa, sizeof(sa))< 0) {
            //fprintf(stderr, "Failed to connect to remote host: host not there. Now trying to host new chat
session\n");
            if (close(sd) < 0)
                    perror("close");
            sd = server_init(&sa_server);
            tv.tv_sec = 3;
            tv.tv_usec = 0;
            FD_SET(sd,&rfds);
            len = sizeof(struct sockaddr_in);
            if (select(sd+1,&rfds,NULL,NULL,&tv)<0) {
                perror("select");
                exit(1);
            }
            if (FD_ISSET(sd,&rfds)) {
                if (( newsd = accept(sd, (struct sockaddr *)&sa_server, &len)) < 0)
```

```
                perror("accept");
                    else goto hosting;
                }
                else {
                    //fprintf(stderr, "Server has no clients. Trying again as a client\n");
                    if (close(sd) < 0)
                        perror("close");
                    sd=client_init(&sa,port,hp);
                }
            }
            else break;
        }
        fprintf(stderr, "Connected as client.\n");
```

Αρχικά δοκιμάζουμε να συνδεθούμε ως client μέσω της κλήσης connect. Αν η connect επιστρέψει επιτυχώς τότε εκτελείται η break και η ροή του προγράμματος συνεχίζει όπως θα φανεί παρακάτω.
Αν η connect αποτύχει τότε κλείνουμε το συγκεκριμένο fd ο οποίος αναφέρεται σε socket με client configuration και παίρνουμε έναν νέο fd με server configuration καλώντας την server_init.
Αν η μακροεντολή FD_ISSET επιστρέψει true, αυτό σημαίνει οτι ο file descriptor περιέχει δεδομένα άρα μπορεί να κληθεί η accept χωρίς να μπλοκάρει. Η accept επιστρέφει ένα καινούργιο file descriptor στον οποίο μπορούμε πλεον να εκτελέσουμε κλήσεις read και write για να επικοινωνήσουμε με το άλλο μέλος του chat.
Αν η μακροεντολή FD_ISSET επιστρέψει false, τότε δεν υπήρχε καμία αίτηση για νεα σύνδεση για 3 δευτερόλεπτα οπότε κλείνουμε το fd που αντιστοιχίζεται στο server configuration, δημιουργούμε ένα socket με client configuration και ελέγχουμε πάλι αν μπορούμε να συνδεθούμε ως client.
Το ακόλουθο κομμάτι κώδικα εκτελείται σε περίπτωση που η connect επιστρέψει χωρίς σφάλμα, οπότε έχουμε συνδεθεί ως client:

```
for (;;) {
        tv.tv_sec = 0;
        tv.tv_usec = 0;
        FD_SET(sd,&rfds);
        FD_SET(0,&rfds);
        if (select(sd + 1,&rfds, NULL, NULL, &tv) < 0)
            perror("Select\n");
        if (FD_ISSET(sd,&rfds)) {
            n = read(sd, buf, sizeof(buf));
            if (n <= 0) {
                if (n < 0)
                    perror("read from remote peer failed");
                else
                    fprintf(stderr, "Peer went away\n");
                break;
            }
            if (insist_write(1, buf, n) != n) {
                perror("write to remote peer failed");
                break;
            }

        }
        if (FD_ISSET(0,&rfds)) {
```

```
                n = read(0,buf,sizeof(buf));
                if (insist_write(sd, buf, n) != n) {
                        perror("write to remote peer failed");
                        break;
                }
        }
    }
    if (close(sd) < 0)
                perror("close");
    fprintf(stderr, "\nDone.\n");
    return 0;
```

Για την αποφυγή μπλοκαρίσματος απο την κλήση της read χρησιμοποιούμε και πάλι το system call select. Οι μόνες διαφορές στη χρήση της select σε σχέση με προηγουμένως είναι στο ότι πλεον προσθέτουμε και τον file descriptor 0 (stdin) στο σύνολο των υπο εξέταση fd και επίσης η τιμή του πεδίου tv_usec της δομής timeval είναι μηδενισμένη.

Τέλος, παρατίθενται το κομμάτι του κώδικα που εκτελείται όταν πραγματοποιηθεί εγκατάσταση σύνδεσης με διάταξη server

```
hosting:
    fprintf(stderr,"Connected as host\n");
    if (!inet_ntop(AF_INET, &sa_server.sin_addr, addrstr, sizeof(addrstr))) {
                perror("could not format IP address");
                exit(1);
    }
    fprintf(stderr, "Incoming connection from %s:%d\n",addrstr, ntohs(sa_server.sin_port));
    FD_ZERO(&rfds);
    for (;;) {
        tv.tv_sec = 0;
        tv.tv_usec = 0;
        FD_SET(newsd,&rfds);
        FD_SET(0,&rfds);
        if (select(newsd + 1,&rfds, NULL, NULL, &tv) < 0)
                perror("Select\n");
        if (FD_ISSET(newsd,&rfds)) {
                n = read(newsd, buf, sizeof(buf));
                if (n <= 0) {
                        if (n < 0)
                                perror("read from remote peer failed");
                        else
                                fprintf(stderr, "Peer went away\n");
                        break;
                }
                insist_write(1,buf,n);
        }
        if (FD_ISSET(0,&rfds)) {
                n = read(0,buf,sizeof(buf));
                if (insist_write(newsd, buf, n) != n) {
                        perror("write to remote peer failed");
                        break;
                }
        }
    }
    if (close(newsd) < 0)
        perror("close");
    return 0;
```

Όπως μπορούμε να παρατηρήσουμε ο κώδικας είναι ίδιος με προηγουμένως με τη μόνη διαφορά οτι πλεον το file descriptor το οποίο αντιστοιχει στο socket σύνδεσης με τον άλλον peer είναι το newsd, δηλαδή το file descriptor που επέστρεψε η κλήση της accept.

Ζητούμενο 2ο):

Στο δεύτερο κομμάτι της άσκησης μας ζητείται να επεκτείνουμε την υλοποίηση του chat ώστε να υποστηρίζει κρυπτογραφημένη επικοινωνία. Η κρυπτογράφηση θα επιτυγχάνεται με κατάλληλες κλήσεις ioctl σε μια κρυπτογραφική συσκευή που αντιστοιχεί σε πραγματικό hardware. Στην πραγματικότητα, οι κλήσεις ioctl δεν θα πραγματοποιούνται σε πραγματική κρυπτογραφική συσκευή αλλά σε εικονική συσκευή που έχει συνδεθεί στο μηχάνημα μέσω εισαγωγής στον πυρήνα του linux του module cryptodev. Μόλις εισαγθεί επιτυχώς το module, δημιουργείται η εικονική συσκευή crypto η οποία εμφανίζεται στο directory /dev του linux και στην οποίο μπορούμε πλέον να εκτελέσουμε κλήσεις open, read, write, ioctl κλπ. Στην υλοποίησή μας θα χρησιμοποιήσουμε μόνο τις κλήσεις open και ioctl για να επιτύχουμε κρυπτογράφηση δεδομένων. Στη συνέχεια επεξηγούμε τον τρόπο λειτουργίας της εικονικής κρυπτογραφικής συσκευής.

Περιγραφή υλοποίησης

Η υποστήριξη κρυπτογράφησης για την εφαρμογή chat προϋποθέτει τη χρήση ενός προσυμφωνημένου μυστικού κλειδιού που θα χρησιμοποιούν και τα δυο μέλη της συζήτησης για την κρυπτογράφηση και αποκρυπτογράφηση των δεδομένων. Στην υλοποίησή μας υποθέτουμε οτι και τα δυο μέλη έχουν συμφωνήσει πριν την εγκατάσταση σύνδεσης για ένα κοινό μυστικό κλειδί. Συγκεκριμένα, για λόγους ευκολίας έχουμε επιλέξει το κλειδί 0123...15 αλλά θα μπορούσαμε να είχαμε επιλέξει οποιαδήποτε πληροφορία μπορεί να γνώριζαν μόνο τα δυο μέλη. Η επιλογή αυτή προσφέρει το πλεονέκτημα ότι ένα πιθανός "παρατηρητής" της συνομιλίας δεν θα μπορεί να γνωρίζει το μυστικό κλειδί και θα χρειαστεί μεθόδους brute force για να αποκρυπτογραφήσει τη συνομιλία αλλά δεν προσφέρει ευελιξία στην επιλογή του κλειδιού. Εφόσον ο κακόβουλος παρατηρητής σπάσει το μυστικό κλειδί, θα μπορεί να αποκρυπτογραφήσει όλες τις συνομιλίες των δυο συνομιλητών στο μέλλον. Μια δεύτερη προσέγγιση στην επιλογή του μυστικού κλειδιού θα ήταν τα δυο μέλη να συμφωνούν σε ένα κοινό κλειδί κατά την εγκατάσταση της σύνδεσης. Αυτή η προσέγγιση προσφέρει μεγαλύτερη ευελιξία στην επιλογή του κλειδιού αλλά επίσης και λιγότερη ασφάλεια διότι ο "παρατηρητής" θα μπορούσε να μάθει το συμφωνημένο κλειδί αν παρατηρούσε όλη την κίνηση του δικτύου απο την αρχή της συνεδρίας. Μια πιθανή βελτίωση σε αυτή την προσέγγιση είναι η χρήση κρυπτογράφησης δημόσιου κλειδιού RSA για την κρυπτογράφηση του μυστικού κλειδιού AES κατά τη διαδικασία συμφώνησής του. Έτσι ο "παρατηρητής" ακόμα και αν γνωρίζει όλη την κίνηση του δικτύου δεν θα μπορέσει να μάθει το μυστικό κλειδί διότι ο αλγόριθμος RSA προσφέρει μέγιστη ποιότητα ασφάλειας και είναι πολύ ανθεκτικός σε brute force επιθέσεις.

Λεπτομέρειες υλοποίησής

Η προσθήκη υποστήριξης για κρυπτογράφηση στην υπάρχουσα υποδομή για την εφαρμογή chat δεν απαιτεί πολλές τροποποιήσεις στον κώδικα του πρώτου ζητήματος. Συγκεκριμένα, απαιτείται η προσθήκη δηλώσεων των μεταβλητών που χρησιμοποιούνται, το άνοιγμα της κρυπτογραφικής συσκευή και οι κλήσεις ioctl πριν την αποστολή και μετά τη λήψη δεδομένων.

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

        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;
        /*Opening new cryptodev file descriptor*/
        cfd = open("/dev/crypto", O_RDWR);
        if (cfd < 0) {
                perror("open(/dev/crypto)");
                return 1;
        }

        /*Setting up basic crypting session parameters*/
        memset(&sess, 0, sizeof(sess));
        memset(&cryp, 0, sizeof(cryp));
        for (i=0;i<BLOCK_SIZE;i++) data.iv[i]='0';
        for (i=0;i<KEY_SIZE;i++) data.key[i]=i;
        sess.cipher = CRYPTO_AES_CBC;
        sess.keylen = sizeof(data.key);
        sess.key = data.key;
        if (ioctl(cfd, CIOCGSESSION, &sess)) {
                perror("ioctl(CIOCGSESSION)");
                return 1;
        }
        cryp.ses = sess.ses;
        cryp.len = sizeof(data.in);
        cryp.iv = data.iv;
```

Ακολούθως δίνεται το κομμάτι κώδικα που εκτελείται όταν ο peer έχει συνδεθεί ως client. Στην περίπτωση σύνδεσης ως server, δεν υπάρχει διαφορά, παρα μόνο οτι η κλήσεις read/write γίνονται στον newsd, γι'αυτό και αυτό το κομμάτι δεν δίνεται.

```
for (;;) {
                tv.tv_sec = 0;
                tv.tv_usec = 0;
```

```
        FD_SET(sd,&rfds);
        FD_SET(0,&rfds);
        if (select(sd + 1,&rfds, NULL, NULL, &tv) < 0)
                perror("Select\n");
        if (FD_ISSET(sd,&rfds)) {
                n = read(sd, data.encrypted, sizeof(data.encrypted));
                if (n <= 0) {
                        if (n < 0)
                                perror("read from remote peer failed");
                        else
                                fprintf(stderr, "Peer went away\n");
                        break;
                }
                cryp.op = COP_DECRYPT;
                cryp.src = data.encrypted;
                cryp.dst = data.decrypted;
                if (ioctl(cfd, CIOCCRYPT, &cryp)) {
                        perror("ioctl(CIOCCRYPT)");
                        return 1;
                }
                i=0;
                while (data.decrypted[i]!='\n') i++;
                if (insist_write(1, data.decrypted, i+1) != i+1) {
                        perror("write to remote peer failed");
                        break;
                }

        }
        if (FD_ISSET(0,&rfds)) {
                n = read(0,data.in,sizeof(data.in));
                cryp.op = COP_ENCRYPT;
                cryp.src = data.in;
                cryp.dst = data.encrypted;
                if (ioctl(cfd, CIOCCRYPT, &cryp)) {
                        perror("ioctl(CIOCCRYPT1)");
                        return 1;
                }
                        if (insist_write(sd, data.encrypted, sizeof(data.encrypted)) != sizeof(data.encrypted))
{
                        perror("write to remote peer failed");
                        break;
                }
        }
    }
    fprintf(stderr, "\nDone.\n");
    goto out;
```

Μόλις η μακροεντολή FD_ISSET(sd,&rfds) γίνει true, δηλαδή η διαδικασία λάβει δεδομένα απο το socket που έχει συνδεθεί με τον άλλο συνομιλητή, εκτελείται η read και τα δεδομένα γράφονται στον πίνακα data.encrypted διότι έχουν υποστεί κρυπτογράφηση απο τον αποστολέα. Κατόπιν ορίζουμε να εκτελεστεί λειτουργία αποκρυπτογράφησης μέσω της εντολής cryp.op=COP_DECRYPT και στη συνέχεια, μέσω της cryp.src ενημερώνουμε τη συσκευή που θα βρει το πρώτο byte δεδομένων. Με την εντολή cryp.dst ενημερώνουμε τη συσκευή που να τοποθετήσει τα αποκρυπτογραφημένα δεδομένα. Τέλος καλούμε την ioctl(cfd,CIOCCRYPT,&cryp) και μόλις αυτή εκτελεστεί επιτυχώς, γράφουμε τα αποκρυπτογραφημένα δεδομένα στο stdout.

Αντίστοιχα, μόλις η FD_ISSET(0,&rfds) γίνει true, δηλαδή η διαδικασία λάβει δεδομένα απο τον χρήστη, εκτελείται η read η

οποία εγγράφει τα δεδομένα στο data.in. Στη συνέχεια ενημερώνεται η συσκευή οτι πρόκειται να εκτελεστεί λειτουργία κρυπτογράφησης μέσω της εντολής cryp.op=COP_ENCRYPT, οτι τα προς κρυπτογράφηση δεδομένα βρίσκονται στο data.in και τα κρυπτογραφημένα θέλουμε να εγγραφούν στο data.encrypted Τέλος καλείται η ioctl(cfd,CIOCCRYPT,&cryp), μετά την επιτυχή εκτέλεση της οποίας μπορεί η εφαρμογή να στείλει τα δεδομένα στον άλλο συνομιλητή μέσω της write στο file descriptor που αντιστοιχεί στο socket.

Τέλος φροντίζουμε για το σωστό τερματισμό του session με την κρυπτογραφική συσκευή.

```
out:
    if (ioctl(cfd, CIOCFSESSION, &sess.ses)) {
        perror("ioctl(CIOCFSESSION)");
        return 1;
    }
```

Ζητούμενο 3ο)

Στο τρίτο μέρος της άσκησης μας ζητείται να υλοποιήσουμε μια εικονική συσκευή κρυπτογράφησης που θα βρίσκεται μέσα σε εικονικό μηχάνημα. Ουσιαστικά, αυτό το κομμάτι πρόκειται για επέκταση του δεύτερου ζητήματος της άσκησης. Η εφαρμογή chat θα εκτελείται πλέον σε εικονικό μηχάνημα μέσω της εφαρμογής paravirtualization QEMU-KVM και το εικονικό μηχάνημα θα εξομοιώνει τον πυρήνα του linux. Για διάκριση των δυο μηχανημάτων χρησιμοποιούμε τους όρους host και guest. Το host μηχάνημα θα τρέχει την userspace εφαρμογή QEMU-KVM η οποία δημιουργεί μέσω virtualization ένα εικονικό μηχάνημα linux με έκδοση 3.2.0-0.bpo.4-amd64, το οποίο διακρίνουμε ως guest μηχάνημα. Μέσα στο guest μηχάνημα θα τρέχει το chat με κρυπτογράφηση αλλά επειδή η πραγματική κρυπτογραφική συσκευή βρίσκεται στο host μηχάνημα, το chat δεν μπορεί να πραγματοποιήσει κλήσεις συστήματος προς αυτή. Σκοπός της άσκησης είναι η δημιουργία εικονικής συσκευής κρυπτογράφησης η οποία θα είναι συνδεδεμένη στο guest μηχάνημα. Κλήσεις προς αυτή την εικονική συσκευή θα αντιστοιχίζονται σε κλήσεις προς την πραγματική συσκευή μέσω του προτύπου paravirtualized οδηγών συσκευών VirtIO. Στην ουσία θα υλοποιηθεί ένα οδηγός για την εικονική συσκευή, ο οποίος αποτελείται απο δυο κομμάτια. Το ένα κομμάτι είναι το frontend μέρος του οδηγού, δηλαδή ο κώδικας πυρήνα που θα βρίσκεται στο guest μηχάνημα, και το άλλο κομμάτι είναι το backend το οποίο θα βρίσκεται στον πηγαίο κώδικα του QEMU.

Περιγραφή υλοποίησης

Για να γίνει κατανοητή η υλοποίηση της άσκησης, θα δοθεί αρχικά η πλήρης πορεία για τη σωστή εκτέλεση του οδηγού.

Αρχικά, εκκινούμε το εικονικό περιβάλλον απο το host μηχάνημα τρέχοντας την userspace εφαρμογή QEMU με τον τροποποιημένο κώδικά της ώστε να υποστηρίζει το backend μέρος του οδηγού μας. Υπενθυμίζουμε οτι στο host μηχάνημα έχει εισαχθεί το module cryptodev το οποίο δημιουργεί μια συσκευή κρυπτογράφησης στον κατάλογο /dev με ονομα crypto. Η συσκευή αυτή πρόκειται για εικονοποίηση του hardware μιας πραγματικής κρυπτογραφικής συσκευής αλλά από εδώ και στο εξής θα ανφερόμαστε στην συσκευή /dev/crypto με τον όρο πραγματική κρυπτογραφική συσκευή.

Επίσης, εχουμε ορίσει ως παράμετρο εκκίνησης του guest μια νέα pci συσκευή η οποία θα αντιστοιχεί στην εικονική συσκευή που υλοποιεί την κρυπτογράφηση μέσα στο guest. Αφού εκκινήσει το λειτουργικό σύστημα στο paravirtualized εικονικό μηχάνημα, εισάγουμε στον πυρήνα του το module το οποίο περιέχει το frontend κομμάτι του οδηγού για την pci συσκευή που εισάγαμε κατά την εκκίνηση. Η εισαγωγή του module προκαλεί τη δημιουργία της ουράς VirtQueue η οποία θα είναι υπεύθυνη για την επικονωνία του backend και frontend μέρους του οδηγού. Στη συνέχεια δημιουργούμε τον αντίστοιχο κομβο /dev/cryptodev με το ίδιο major number το οποίο έχουμε δηλώσει και στον οδηγό.

Απο το σημείο αυτό και μετά, μπορούμε να πραγματοποιήσουμε κλήσεις συστήματος open, read, write, ioctl κλπ πάνω σε αυτή τη συσκευή οι οποίες αντιστοιχίζονται μέσω του major number της συσκευή με τις δικές υλοποιήσεις αυτών των κλήσεων στο frontend driver. Εκτελώντας open στη συσκευή /dev/cryptodev προκαλείται η κλήση της open του οδηγού μας, η οποία εχει ως αποτέλεσμα την αποστολή εντολής open για την πραγματική κρυπτογραφική συσκευή /dev/crypto στο host μηχάνημα. Ο host δέχεται την εντολή, εκτελεί την open και ενημερώνει κατάλληλα τον guest μεσω του VirtQueue. Ο guest δέχεται interrupt και επεξεργάζεται τα δεδομένα που του έστειλε ο host. Κλήση της ioctl απο τον guest προκαλεί ανάλογα αποτελέσματα. Συγκεκριμένα, ο guest στέλνει κατάλληλη εντολή στον host μέσω του VirtQueue, ο host δέχεται την εντολή, καλεί την ioctl πάνω στην πραγματική κρυπτογραφική συσκευή και επιστρέφει τα αποτελέσματά της πίσω στον guest.

Η λίστα scatter-gather είναι μια δομή η οποία χρησιμοποιείται για DMA μεταφορές δεδομένων. Επειδή τα δεδομένα προς μεταφορά μπορεί να βρίσκονται διάσπαρτα στη μνήμη, η απόδοση του DMA μπορεί να μειωθεί αρκετά επειδή θα πρέπει ο μηχανισμός DMA να συνεργάζεται συνεχώς με το λειτουργικό για να μαθαίνει τις διάσπαρτες διευθύνσεις μνήμης των δεδομένων προς μεταφορά. Το πρόβλημα αυτό λύνει η λίστα scatter-gather η οποία φροντίζει για τη συλλογή όλων των διάσπαρτων buffer στη μνήμη σε μια κεντρική δομή, όπως φανερώνει και το όνομά της. Έτσι οι λειτουργίες DMA επιταχύνονται σημαντικά.

Λεπτομέρειες υλοποίησής

Ακολούθως θα δοθεί η περιγραφή της άσκησης σύμφωνα με την πορεία μιας συνηθισμένης εκτέλεσης του chat στο guest.

Απο εδω και πέρα θεωρούμε οτι η διαδικασία που περιγράφηκε παραπάνω έχει ήδη πραγματοποιηθεί οπότε έχουν δημιουργηθεί τα απαραίτα devices και virtqueues κατα την εισαγωγή του module.

Στην υλοποίηση μας και τα δυο peers του chat βρίσκονται μέσα στο guest μηχάνημα οπότε για την λειτουργία του chat απαιτούνται δυο εικονικες συσκευές κρυπτογράφηση, μια για κάθε peer. Για τη συγκεκριμένη υλοποίηση, κάθε συσκευή λειτουργεί ανεξάρτητα απο την άλλη οπότε δεν εμφανίζονται συνθήκες ανταγωνισμού μεταξύ των λειτουργιών κρυπτογράφησης απο τα δυο peers. Θα μπορούμε να είχαμε επεκτείνει τη λειτουργία του οδηγού ώστε να υποστηρίζει πολλαπλές λειτουργίες κρυπτογράφησης πάνω στην ίδια εικονική συσκευή μέσω της χρήσης σημαφόρων.

Αρχικά ο κάθε peer πραγματοποιεί μια λειτουργία open πάνω στη συσκευή του (/dev/crytodev0 ,...). Αυτό προκαλεί μια κλήση της crypto_chrdev_open λόγω της αντιστοίχισης που υπάρχει απο τη δομή file_operations του module που έχουμε εισάγει. Αυτή ελέγχει αρχικά αν έχει εισαχθεί το κατάλληλο εικονικό device στο μηχάνημα και αν δεν είναι ήδη ανοικτή η συσκευή βάσει του minor number. Εαν δεν υπάρχει κανένα πρόβλημα στις παραπάνω λειτουργίες, τότε στέλνει ένα μήνημα ελέγχου στο backend μέρος μέσω της χρήσης προκαθορισμένων σταθερών, της event και της value, των οποίων ο συνδυασμών προσδιορίζουν το μήνυμα ελέγχου. Την αποστολή του μηνήματος αναλαμβάνει η συνάρτηση send_control_msg η οποία βάζει στην c_ovq virtqueue τις δυο αυτές τιμές χρησιμοποιώντας μια scatter-gather λίστα. Αυτό το καταφέρνει με τις συναρτήσεις sg_init_one , η οποία αρχικοποιέι την scatter-gather λίστα με τις παραπάνω τιμές , και την virtqueue_add_buf , η οποία τοποθετεί σε μια virtqueue την sg λίστα. Στη συνέχεια , ειδοποιεί το backend μέρος μέσω της virtqueue_kick , και περιμένει επιβεβαίωση από αυτο.

Το backend , εφόσον είναι σε userspace , δεν μπορεί να τρέχει σε interrupt context , οπότε για να ειδοποιήθει από το frontend μέρος παρακολουθεί προκαθορισμένες θέσεις μνήμης για κάθε virtqueue , για να ενημερωθεί για αλλαγές. Όταν γίνει αυτό , καλείται η συνάρτηση που έχει συνδεθεί με το συγκεκριμένο vq κατά την αρχικοποίηση του virtio device , δηλαδή κατά την εκκίνηση του utopia. Στην συγκεκριμένη περίπτωση , καλείται η control_out , η οποία λαμβάνει τις τιμές της sg λίστας , και τις περνάει στην συνάρτηση handle_control_message σε μορφή buffer. Αυτή εξετάζει τον συνδυασμό των event και value , και εκτελεί την επιθυμητή λειτουργία στην πραγματική συσκευή κρυπτογράφησης. Συγκεκριμένα , ο κώδικάς της handle_control_message είναι ο εξής :

```
static void handle_control_message(VirtIOCrypto *crdev, void *buf, size_t len)
{
    struct virtio_crypto_control cpkt, *gcpkt;

    FUNC_IN;
    gcpkt = buf;

    if (len < sizeof(cpkt)) {
        /* The guest sent an invalid control packet */
        return;
```

```
    }

    cpkt.event = lduw_p(&gcpkt->event);
    cpkt.value = lduw_p(&gcpkt->value);

    if (cpkt.event == VIRTIO_CRYPTO_DEVICE_GUEST_OPEN) {
        /* cpkt.value = 1 for file open and 0 for file close. */
        if (cpkt.value) {
            printf("in open file\n");

            /* Open crypto device file and send the appropriate
             * message (event) to the guest */
            /* ? */
            crdev->fd = open("/dev/crypto",O_RDWR);
            printf("open fd=%d\n",crdev->fd);
            send_control_event(crdev,VIRTIO_CRYPTO_DEVICE_HOST_OPEN,crdev->fd);
        }
        else {
            printf("in close file\n");

            /* Close the previously opened file */
            /* ? */
            if(close(crdev->fd)<0)
send_control_event(crdev,VIRTIO_CRYPTO_DEVICE_HOST_CLOSE,-1);    //TODO Needs close error
checking
            else send_control_event(crdev,VIRTIO_CRYPTO_DEVICE_HOST_CLOSE,0);
        }
    }
    FUNC_OUT;
}
```

Αυτή καλεί την συνάρτηση send_control_event , η οποία με τον ίδιο τρόπο , αλλά στην c_ivq , ενημερώνει το frontend μέρος μέσω κλήσης της virtqueue_notify πάνω στην c_ivq, για την επιτυχία ή όχι της επιθυμητής λειτουργίας. Ταυτόχρονα , στην περίπτωση της open , επιστρέφει και την τιμή του file descriptor. Η virtqueue_notify, σε αντίθεση με την kick, προκαλεί interrupt στο frontend μέρος. Τέλος, η control_out καλεί την virtqueue_notify στην c_ovq και τοποθετεί ένα κενό buffer στην ίδια για να γνωστοποιήσει το frontend μέρος ότι έχει επεξεργαστεί το μήνυμα. Άν έχουμε επιλέξει να είναι blocking η open, τότε το frontend μέρος μπλοκάρει μέσω της κλήσης wait_event_interruptible(crdev->c_wq,crypto_device_ready(crdev));
η οποίο μπλοκάρει μέχρι να γίνει wake up απο την handle_control_message μόνο όταν η πραγματική συσκευή είναι ανοιχτή.

Σε αυτό το σημείο, αν όλα έχουν πάει καλά, έχει πραγματοποιηθεί επιτυχώς το άνοιγμα της πραγματικής κρυπτογραφικής συσκευής στο host μηχάνημα και είναι έτοιμη για την εκτέλεση των λειτουργιών ioctl.

Στη συνέχεια, κατά την εκτέλεση του chat πραγματοποιείται κλήση της ioctl στον fd που επέστρεψε η προηγούμενη open οπότε καλείται η crypto_chrdev_ioctl (file operations) η οποία καλεί την crypto_ioctl με παραμέτρους την δομή file που προκάλεσε την ioctl, την εντολή που θέλουμε να εκτελέσει και τα arguments της εντολής.

Πέρα απο τις βασικές αρχικοποιήσεις, η συνάρτηση αυτή ελέγχει τη μεταβλητή cmd και πραγματοποιεί τις αντίστοιχες ενέργειες. Αρχικά δημιουργεί την δομή crdata η οποία περιέχει το σύνολο των

δεδομένων που χρειάζεται η ioctl της πραγματικής κρυπτογραφικής συσκευής που βρίσκεται στον host. Διακρίνουμε τρεις περιπτώσεις ioctl ανάλογα με τη μεταβλητή cmd. Στην πρώτη περίπτωση, δηλαδή όταν το cmd είναι CIOCGSESSION, τοποθετούμε στην δομή crdata->op.sess τα δεδομένα που έστειλε ο χώρος χρήστη μέσω της κλήσης στην copy_from_user και αυτά τα δεδομένα τα στέλνουμε στο backend μέρος καλώντας την συνάρτηση send_buf. Αυτή αναλαμβάνει την αποστολή των δεδομένων σε μορφή sg list πραγματοποιώντας παρόμοιες εντολές με την send_control_msg που αναφέρθηκε παραπάνω, με τη διαφορά οτι τα δεδομένα εγγράφονται πλεον στην ovq και όχι στην c_ovq και είτε μπλοκάρει περιμένοντας την επεξεργασία των δεδομένων απο το backend μέρος είτε τερματίζει κανονικά αναλόγως της επιλογής μας για την υλοποίηση. Εμεις επιλέξαμε να μπλοκάρει θέτοντας τη μεταβλητή nonblock false. Τέλος ειδοποιεί το backend μέρος μέσω της κλήσης virtqueue_kick. Στη συνέχεια μπλοκάρει μέσω της κλήσης στην wait_event_interruptible(crdev->i_wq,device_has_data(crdev));
περιμένοντας την απάντηση απο τον host.

Στο backend μέρος, αρχικά καλείται η handle_output η οποία αναλαμβάνει να επεξεργαστεί κατάλληλα την sg λίστα και να τοποθετήθει τα δεδομένα σε buffer τον οποίο θα επεξεργαστεί η crypto_handle_ioctl_packet. Αυτή ελέγχει ποιά εντολή έχουμε στείλει για εκτέλεση , και την πραγματοποιεί με τα δεδομένα που της έχουμε στείλει.Στην συνέχεια , στέλνει την επεξεργασμένη ,από την κρυπτογραφική συσκευή, δομή cr_data πίσω στο frontend μέσω της συνάρτησης send_buffer. Εκεί επίσης κάνουμε notify to frontend στην ivq. Ο κώδικας της crypto_handle_ioctl_packet είναι ο εξής:

```
switch (cr_data->cmd) {

    /* set the metadata for every operation and perform the ioctl to
     * the actual device */
    case CIOCGSESSION:
        /* ? */
        cr_data->op.sess.key=cr_data->keyp;
        if (ioctl(crdev->fd, CIOCGSESSION, &cr_data->op.sess))
            perror("ioctl(CIOCGSESSION)");                    //TODO ERROR CHECKING
        cr_data->op.sess_id = cr_data->op.sess.ses;
        break;

    case CIOCCRYPT:
        cr_data->op.crypt.src=cr_data->srcp;
        cr_data->op.crypt.dst=cr_data->dstp;
        cr_data->op.crypt.iv=cr_data->ivp;
        if (ioctl(crdev->fd, CIOCCRYPT, &cr_data->op.crypt))
            perror("ioctl(CIOCCRYPT)");
        /* ? */
        break;

    case CIOCFSESSION: );
        if (ioctl(crdev->fd, CIOCFSESSION, &cr_data->op.sess.ses))
            perror("ioctl(CIOCFSESSION)");
        /* ? */

        break;

    default:
        printf("BAD CMD\n");
```

```
        return -EINVAL;
    }
```

Όταν ξυπνήσει το frontend μέρος, επιστρέφει τα δεδομένα στο userspace με την copy_to_user και ελευθερώνει την μνήμη που δεσμεύει η δομή cr_data , καθώς σε καθέ λειτουργία την ξαναδημιουργούμε.

Η διαδικασία για τις άλλες 2 λειτουργίες της crypto device (CIOCCRYPT και CIOCFSESSION) είναι παρόμοιες με την παραπάνω ,μόνο που πρέπει να φροντίσουμε , για την σωστη μεταφορά των δεδομένων καθώς η δομή cr_data->op.crypt περιλαμβάνει δείκτες σε δεδομένα και όχι τα ίδια , οπότε και δεν μπορεί να γίνει η σωστή μεταφόρα τους. Αυτό επιτυγχάνεται με την χρήση πινάκων μέσα στην δομή cr_data, και την εγγραφή των ζητουμένων δεδομένων. Ωστόσο , καθώς δεν γνωρίζουμε το μέγεθος του iv στο userspace , έχουμε ορίσει μια συγκεκριμένη τιμή ψηφίων. Οπότε όταν το μέγεθος του πίνακα iv στο userspace είναι μικρότερο από αυτήν την τιμή θα μεταφερθούν περιττά δεδομένα.

Στην υλοποιήση μας της εικονικής κρυπτογραφικής συσκευής, υποστηρίζονται μόνο οι 3 από τις πιθανές λειτουργιές της πραγματικης συσκευής.

Παρακάτω δίδεται το τμήμα κώδικά για την υλοποίηση της CIOCRYPT στο frontend.

...

```
case CIOCCRYPT:
        /* ? */

        ret=copy_from_user(&cr_data->op.crypt,(void __user *)arg,sizeof(struct crypt_op));
        ptr_to_dst = cr_data->op.crypt.dst;
        ptr_to_iv = cr_data->op.crypt.iv;
        memcpy(cr_data->srcp,cr_data->op.crypt.src,cr_data->op.crypt.len);
      memcpy(cr_data->dstp,cr_data->op.crypt.dst,cr_data->op.crypt.len);
            for(i=0;i<sizeof(cr_data->ivp);i++)         cr_data->ivp[i]=cr_data->op.crypt.iv[i];
            send_buf(crdev,cr_data,sizeof(struct crypto_data),false);

            if (!device_has_data(crdev)) {
                    printk(KERN_WARNING "sleeping in CIOCCRYPTO\n");
            if (filp->f_flags & O_NONBLOCK)
            return -EAGAIN;

            /* Go to sleep until we have data. */
            ret = wait_event_interruptible(crdev->i_wq,                 device_has_data(crdev));

            if (ret < 0)
                    goto free_buf;
        }

        ret = fill_readbuf(crdev, (char *)cr_data, sizeof(crypto_data));
        memcpy(ptr_to_dst,cr_data->dstp,cr_data->op.crypt.len);
        cr_data->op.crypt.iv=ptr_to_iv;
        ret = copy_to_user((void __user *)arg, &cr_data->op.crypt,
                    sizeof(struct crypt_op));
        if (ret)
```

```
                goto free_buf;

            /* copy the response to userspace */
            /* ? */

            break;

...

default:
            return -EINVAL;
        }


free_buf:
        kfree(cr_data);
        return ret;
}
```