

TABLE DES MATIÈRES

Table des matières	1
Introduction	3
I Problématique	5
1 Introduction	7
2 Systèmes d'exploitation	9
2.1 Rôle d'un système d'exploitation	9
2.2 Architecture Intel	10
2.3 Cas de Linux	14
3 Cas d'étude	17
3.1 Description du problème	19
3.2 Principes de l'analyse	19
3.3 Implantation	21
3.4 Conclusion	21
4 État de l'art	23
II Analyse statique par typage	25
5 Typage	27
5.1 Présentation et but	27
5.2 Taxonomie	27
5.3 Exemples	31
6 Un premier système de types	33
6.1 But	33
6.2 Chaîne de compilation	33
6.3 Syntaxe	34
6.4 Sémantique (opérationnelle, à petits pas)	35
6.5 Règles de typage	38
6.6 Limitations	41
7 Système avec le qualificateur "user"	43
7.1 Éditions et ajouts	43
7.2 Passage sur le cas d'étude	43
7.3 Propriété d'isolation mémoire	43

8	Système avec le qualificateur "sz"	45
8.1	But	45
8.2	Annotation de <code>string.h</code>	45
9	Implantation	47
	Conclusion	49
	Liste des figures	51
	Table des figures	51
	Bibliographie	53
	Bibliographie	55

INTRODUCTION

Première partie

Problématique

INTRODUCTION

SYSTÈMES D'EXPLOITATION

Le système d'exploitation est le programme qui permet à un système d'informatique d'exécuter d'autres programmes. Son rôle est donc capital et ses responsabilités multiples. Dans ce chapitre, nous allons voir quel est son rôle, et comment il peut être implémenté. Pour ce faire, nous étudierons l'exemple d'une architecture Intel 32 bits, et d'un noyau Linux 2.6.

Pour une description plus détaillée des rôles d'un système d'exploitation ainsi que plusieurs cas d'étude détaillés, on pourra se référer à [Tan07].

2.1 Rôle d'un système d'exploitation

Un ordinateur est constitué de nombreux composants matériels : micro-processeur, mémoire, et divers périphériques. Pourtant, au niveau de l'utilisateur, de dizaines de logiciels permettant d'effectuer toutes sortes de calculs et de communication. Le système d'exploitation permet de faire l'interface entre ces niveaux d'abstraction.

Au cours de l'histoire des systèmes informatiques, la manière de les programmer a beaucoup évolué. Au départ, les programmeurs avaient accès au matériel dans son intégralité : toute la mémoire pouvait être accédée, toutes les instructions pouvaient être utilisées.

Néanmoins c'est un peu restrictif : si on est seul à utiliser un système, on est par définition... seul à pouvoir l'utiliser. Dans la seconde moitié des années 60, sont apparus les premiers systèmes "à temps partagé", permettant à plusieurs utilisateurs de travailler en même temps.

Permettre l'exécution de plusieurs programmes en même temps est une idée révolutionnaire, mais elle n'est pas sans difficultés techniques : en effet les ressources de la machine doivent être aussi partagées entre les utilisateurs et les programmes. Par exemple, plusieurs programmes vont par exemple utiliser le CPU les uns à la suite des autres (partage *temporel*) ; et chaque programme aura à sa disposition une partie de la mémoire principale, ou du disque dur (partage *spatial*).

Si deux programmes (ou plus) s'exécutent de manière concurrente sur le même matériel, il faut s'assurer par exemple que les deux s'exécutent à peu près aussi souvent, ou que l'un ne puisse pas écrire dans la mémoire de l'autre. Ce sont des rôles du système d'exploitation.

Cela passe donc par un certain bridage des possibilités du programme : plutôt que de le faire exécuter n'importe quel type d'instruction, il communique avec le système d'exploitation. Bien que cela ait l'air d'une limitation, c'est aussi bénéfique pour le programmeur puisque cela permet de définir des abstractions au niveau du noyau.

Par exemple, si un programmeur veut copier des données depuis un CD-ROM vers la mémoire principale, il devra interroger le bus SATA, interroger le lecteur sur la présence d'un disque dans le lecteur, activer le moteur, calculer le numéro de trame des données sur le disque, demander la lecture, puis déclencher une copie de la mémoire.

Si dans un autre cas il voulait récupérer des données depuis une mémoire USB, il devrait interroger le bus USB, rechercher le bon numéro de périphérique, le bon numéro de canal dans celui-ci, lui appliquer une commande de lecture au bon numéro de bloc, puis copier la mémoire.

Ces deux opérations, bien qu'elles aient le même but (copier de la mémoire depuis un périphérique amovible), ne sont pas effectuées en pratique de la même manière. C'est pourquoi le système d'exploitation fournit les notions de fichier, lecteur, etc : le programmeur n'a plus qu'à utiliser des commandes de haut niveau ("monter un lecteur", "ouvrir un fichier", "lire dans un fichier") et selon le type de lecteur, le système d'exploitation effectuera les actions appropriées.

En résumé, un système d'exploitation est l'intermédiaire entre le logiciel et le matériel, et en particulier assure les rôles suivants :

- Gestion des processus : un système d'exploitation peut permettre d'exécuter plusieurs programmes à la fois. Il faut alors orchestrer ces différents processus et les séparer en terme de temps et de ressources partagées.
- Gestion de la mémoire : chaque processus, en plus du noyau, doit disposer d'un espace mémoire différent. C'est-à-dire qu'un processus ne doit pas pouvoir interférer avec un autre.
- Gestion des périphériques : le noyau étant le seul code à s'exécuter en mode privilégié, c'est lui qui doit communiquer avec les périphériques matériels.
- Abstractions : le noyau fournit aux programmes une interface unifiée, permettant de stocker des informations de la même manière sur un disque dur ou une clef USB (alors que l'accès se déroulera de manière très différente en pratique). C'est ici que la notion arbitraire de fichier sera définie, par exemple.

2.2 Architecture Intel

L'implantation d'un système d'exploitation est très proche du matériel sur laquelle elle va s'exécuter. Pour étudier une implantation en particulier, voyons ce que permet le matériel lui-même.

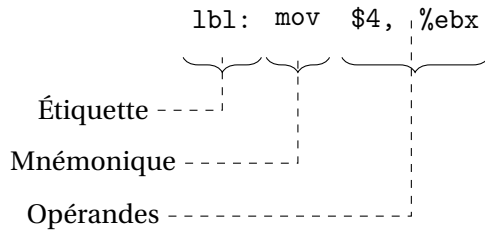
Dans cette section nous décrivons le fonctionnement d'un processeur utilisant une architecture Intel 32 bits. Les exemples de code seront écrits en syntaxe AT&T, celle que comprend l'assembleur GNU.

Assembleur

Pour faire des calculs, le processeur est composé de registres, qui sont des petites zones de mémoire interne, et peut accéder à la mémoire principale.

Les registres principaux sont nommés EAX, EBX, ECX, EDX, ESI et EDI. Ils peuvent être utilisés pour n'importe quel type d'opération, mais certains sont spécialisés : par exemple il est plus efficace d'utiliser EAX en accumulateur, ou ECX en compteur. ESP est le pointeur de pile (*stack pointer*), qui a un rôle particulier dans les instructions push et pop. EBP sert de point de repère dans les appels de fonction. EIP contient l'adresse de l'instruction courante.

Les calculs sont décrits sous forme d'une suite d'instructions. Chaque instruction est composée d'une mnémonique et d'une liste d'opérandes. Les mnémoniques (mov, call, sub, etc) définissent un type d'opération à appliquer sur les opérandes. Elle peut aussi être précédée d'une étiquette, qui correspondra à l'adresse de cette instruction.



Ces opérandes peuvent être de plusieurs types :

- un nombre, noté `$4`
- le nom d'un registre, noté `%eax`
- une opérande mémoire, c'est à dire le contenu de la mémoire à une adresse effective. Cette adresse effective peut être exprimée de plusieurs manières :
 - directement : `addr`
 - indirectement : `(%ecx)`. L'adresse effective est le contenu du registre.
 - "base + déplacement" : `4(%ecx)`. L'adresse effective est le contenu du registre plus le déplacement (4 ici).

En pratique il y a des modes d'adressage plus complexes, et toutes les combinaisons ne sont pas possibles, mais ceux-ci suffiront à décrire les exemples suivants :

- `mov src, dst` copie le contenu de `src` dans `dst`.
- `add src, dst` calcule la somme des contenus de `src` et `dst` et place ce résultat dans `dst`.
- `push src` place `src` sur la pile, c'est à dire que cette instruction décrémente le pointeur de pile `ESP` de la taille de `src`, puis place `src` à l'adresse mémoire de la nouvelle valeur `ESP`.
- `pop src` réalise l'opération inverse : elle charge le contenu de la mémoire à l'adresse `ESP` dans `src` puis incrémente `ESP` de la taille correspondante.
- `jmp addr` saute à l'adresse `addr` : c'est l'équivalent de `mov addr, %eip`.
- `call addr` sert aux appels de fonction : cela revient à `push %eip` puis `jmp addr`.
- `ret` sert à revenir d'une fonction : c'est l'équivalent de `pop %eip`.

Fonctions et conventions d'appel

Dans le langage d'assemblage, il n'y a pas de notion de fonction ; mais `call` et `ret` permettent de sauvegarder et de restaurer une adresse de retour, ce qui permet de faire un saut et revenir à l'adresse initiale. Ce système permet déjà de créer des procédures, c'est-à-dire des fonctions sans arguments ni valeur de retour.

Pour gérer ceux-ci, il faut que les deux morceaux (appelant et appelé) se mettent d'accord sur une convention d'appel commune. La convention utilisée sous GNU/Linux est appelée *cdecl* et possède les caractéristiques suivantes :

- la valeur de retour d'une fonction est stockée dans `EAX`
- `EAX`, `ECX` et `EDX` peuvent être écrasés sans avoir à les sauvegarder
- les arguments sont placés sur la pile (et enlevés) par l'appelant. Les paramètres sont empilés de droite à gauche.

Pour accéder à ses paramètres, une fonction peut donc utiliser un adressage relatif à `ESP`. Cela peut fonctionner, mais cela complique les choses si elle contient aussi des variables locales. En effet, les variables locales sont placées sur la pile, au dessus des (c'est à dire, empilées après) paramètres, augmentant le décalage.

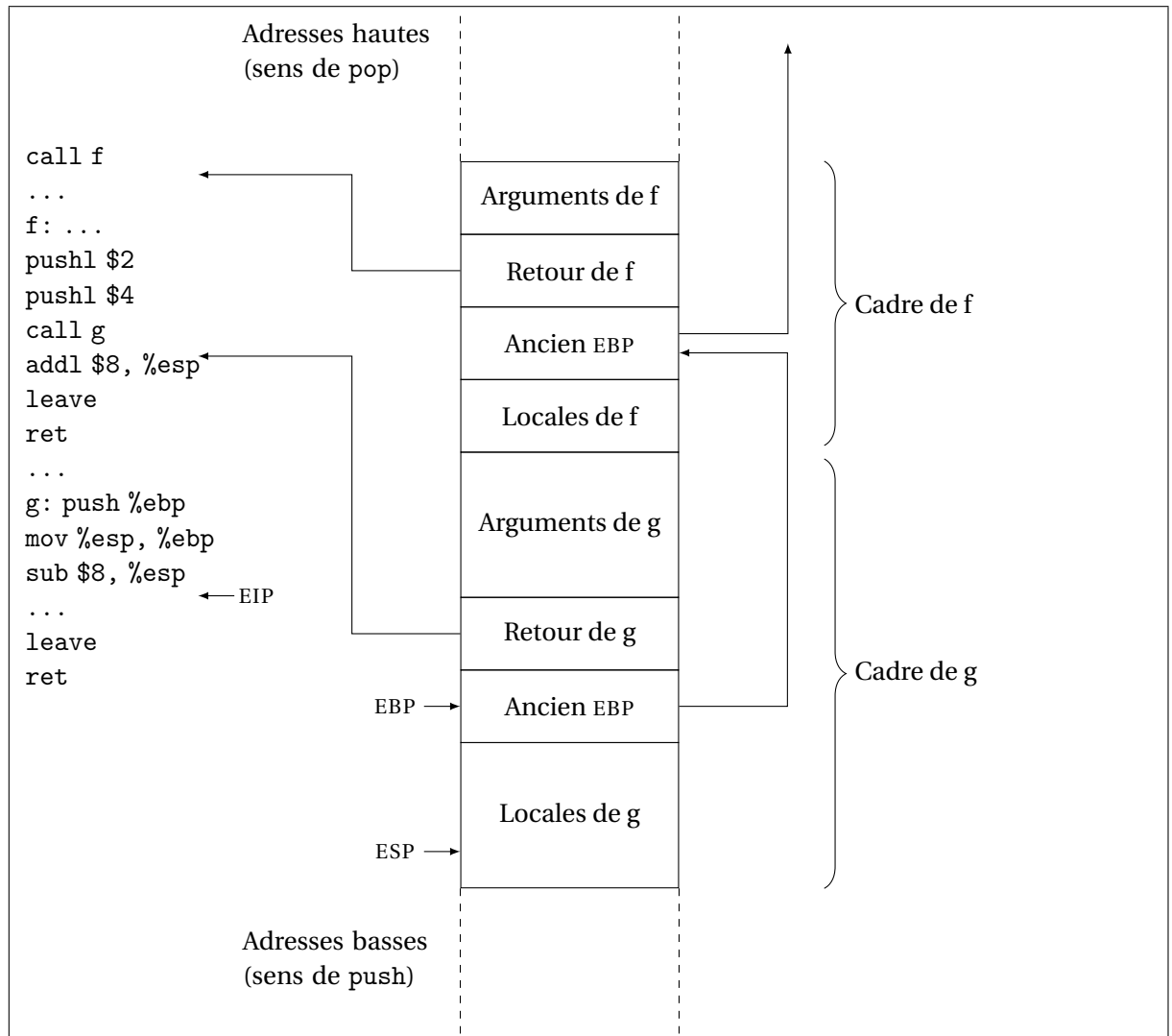


FIGURE 2.1 – Cadres de pile.

Pour simplifier, la pile est organisée en cadres logiques : chaque cadre correspond à un niveau dans la pile d'appels de fonctions. Si `f` appelle `g`, qui appelle `h`, il y aura dans l'ordre sur la pile le cadre de `f`, celui de `g` puis celui de `h`.

Ces cadres sont chaînés à l'aide du registre `EBP` : à tout moment, `EBP` contient l'adresse du cadre de l'appelant.

Prenons exemple sur la figure 2.1 : pour appeler `g(4, 2)`, `f` empile les arguments de droite à gauche. L'instruction `call g` empile l'adresse de l'instruction suivante sur la pile puis saute au début de `g`.

Au début de la fonction, les trois instructions permettent de sauvegarder l'ancienne valeur de `EBP`, faire pointer `EBP` à une endroit fixe dans le cadre de pile, puis allouer 8 octets de mémoire pour les variables locales.

Dans le corps de la fonction `g`, on peut donc se référer aux variables locales par `-4(%ebp)`, `-8(%ebp)`, etc, et aux arguments par `8(%ebp)`, `12(%ebp)`, etc.

À la fin de la fonction, l'instruction `leave` est équivalente à `mov %ebp, %esp` puis `pop %ebp` et



FIGURE 2.2 – Les différents *rings*. Seul le *ring* 0 a accès au hardware via des instructions privilégiées. Pour accéder aux fonctionnalités du noyau, les programmes utilisateur doivent passer par des appels système.

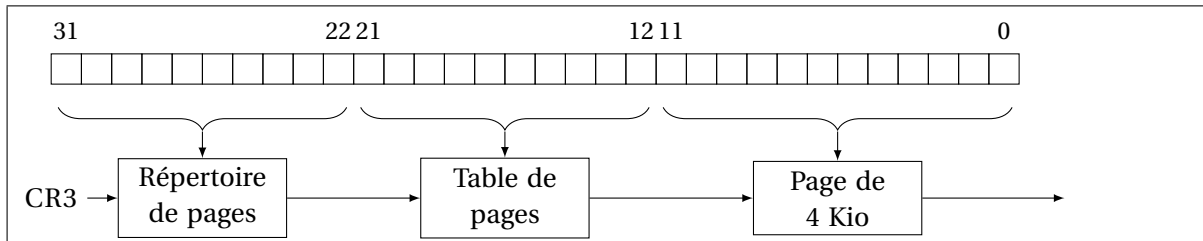


FIGURE 2.3 – Implémentation de la mémoire virtuelle

permet de défaire le cadre de pile, laissant l'adresse de retour en haut de pile. Le `ret` final la dépile et y saute.

Tâches, niveaux de privilèges

Sans mécanisme particulier, le processeur exécuterait uniquement une suite d'instruction à la fois. Pour lui permettre d'exécuter plusieurs tâches, un système de partage du temps existe.

À des intervalles de temps réguliers, le système est programmé pour recevoir une interruption. C'est une condition exceptionnelle (au même titre qu'une division par zéro) qui fait sauter automatiquement le processeur dans une routine de traitement d'interruption. Si à cet endroit le code sauvegarde les registres et restaure un autre ensemble de registres, on peut exécuter plusieurs tâches de manière entrelacée. Si l'alternance est assez rapide, cela peut donner l'illusion que les programmes s'exécutent en parallèle. Comme l'interruption peut survenir à tout moment, on parle de multitâche préemptif.

En plus de cet ordonnancement de processus, l'architecture Intel permet d'affecter des niveaux de privilège à ces tâches, en restreignant le type d'instructions exécutables, ou en donnant un accès limité à la mémoire aux tâches de niveaux moins élevés.

Il y a 4 niveaux de privilèges (nommés aussi *rings* - figure 2.2) : le *ring* 0 est le plus privilégié, le *ring* 3 le moins privilégié. Dans l'exemple précédent, on pourrait isoler l'ordonnanceur de processus en le faisant s'exécuter en *ring* 0 alors que les autres tâches seraient en *ring* 3.

Mémoire virtuelle

À partir du moment où plusieurs processus s'exécutent de manière concurrente, un problème d'isolation se pose : si un processus peut lire dans la mémoire d'un autre, des informations peuvent fuir ; et s'il peut y écrire, il peut en détourner l'exécution.

Le mécanisme de mémoire virtuelle permet de donner à deux tâches une vue différente de la mémoire : c'est à dire que vue de tâches différentes, une adresse contiendra une valeur différente.

defs : intégrité,
confidentialité

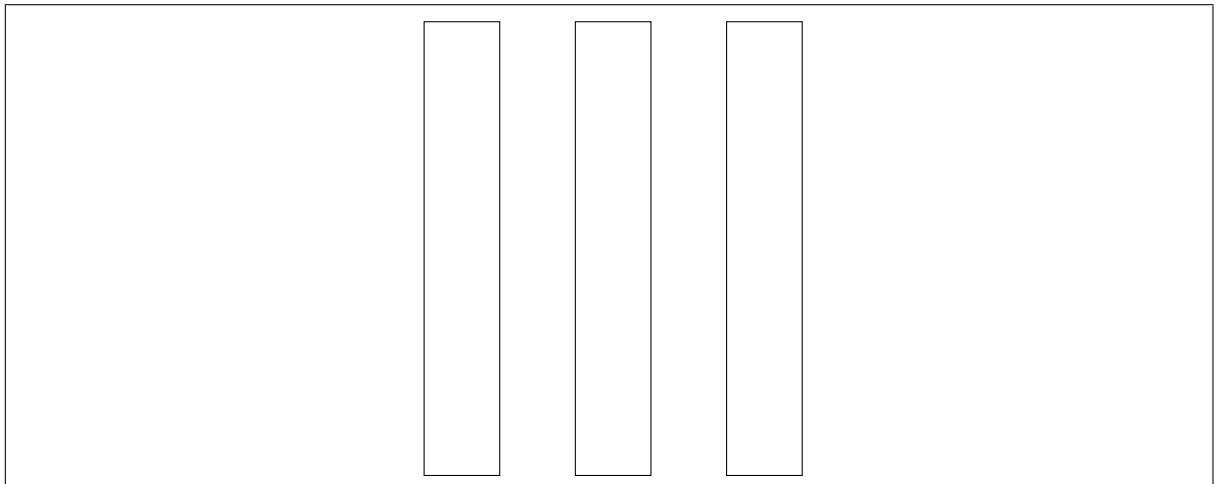


FIGURE 2.4 – Mécanisme de mémoire virtuelle.



FIGURE 2.5 – L'espace d'adressage d'un processus. En gris clair, les zones accessibles à tous les niveaux de privilèges : code du programme, bibliothèques, tas, pile. En gris foncé, la mémoire du noyau, réservée au mode privilégié.

Ce mécanisme est contrôlé par valeur du registre CR3 : les 10 premiers bits d'une adresse sont un index dans le répertoire de pages qui commence à l'adresse contenue dans CR3. À cet index, se trouve l'adresse d'une table de pages. Les 10 bits suivants de l'adresse sont un index dans cette page, donnant l'adresse d'une page de 4 kibi-octets (figure 2.3).

En ce qui concerne la mémoire, les différentes tâches ont une vision différente de la mémoire physique : c'est-à-dire que deux tâches lisant à une même adresse peuvent avoir un résultat différent. C'est le concept de mémoire virtuelle (fig 2.4).

2.3 Cas de Linux

Dans cette section, nous allons voir comment ses mécanismes sont implantés dans le noyau Linux. Une description plus détaillée pourra être trouvée dans [BP05] par exemple.

Deux rings sont utilisés : en *ring* 0, le code noyau et en *ring* 3, le code utilisateur.

Une notion de tâche similaire à celle décrite dans la section 2.2 existe : elles s'exécutent l'une après l'autre, le changement s'effectuant sur interruptions.

Pour faire appel aux services du noyau, le code utilisateur peut faire appel à des appels systèmes, qui sont des fonctions exécutées par le noyau. Chaque tâche doit donc avoir deux piles : une pile "utilisateur" qui sert pour l'application elle-même, et une pile "noyau" qui sert aux appels système.

Appels système

Les programmes utilisateur s'exécutant en *ring* 3, ils ne peuvent pas contenir d'instructions privilégiées, et donc ne peuvent pas accéder directement au matériel (c'était le but!). Pour qu'ils

puissent interagir avec le système (afficher une sortie, écrire sur le disque...), le mécanisme des appels système est nécessaire. Il s'agit d'une interface de haut niveau entre les *rings* 3 et 0. Du point de vue du programmeur, il s'agit d'un ensemble de fonctions C “magiques” qui font appel au système d'exploitation pour effectuer des opérations.

Prenons le cas de l'appel `getpid`, qui retourne le numéro de processus courant. La bibliothèque C fournit une fonction du même nom :

```
pid_t getpid(void);
```

A priori, rien de différent d'une fonction implantée directement en C. Par un processus détaillé ci-après, cette fonction va invoquer la fonction, suivante, définie dans le noyau (`kernel/timer.c`) :

```
SYSCALL_DEFINE0(getpid)
{
    return task_tgid_vnr(current);
}
```

Le mécanisme de couplage entre ces deux fonctions est le suivant. Une explication plus détaillée est disponible dans la documentation d'Intel [Int].

Il y a bien une fonction `getpid` présente dans la bibliothèque C du système. C'est la fonction qui est directement appelée par le programme. Cette fonction commence par placer le numéro de l'appel système (noté `__NR_getpid`, valant 20 ici) dans `EAX`, puis les arguments éventuels dans les registres (`EBX`, `ECX`, `EDX`, `ESI` puis `EDI`). Une interruption logicielle est ensuite déclenchée (int 0x80) : étant donné la configuration du processeur, elle sera traitée en *ring* 0, à un point d'entrée prédéfini (`arch/x86/kernel/entry_32.S`, `ENTRY(system_call)`).

L'exécution reprend donc en *ring* 0, avec une pile propre au processus. Les valeurs des registres ont été préservées : elles sont mises sur la pile du noyau. Le noyau examine le numéro d'appel système (toujours dans `EAX`) et appelle la fonction correspondante dans le tableau `sys_call_table` : les arguments sont en place sur la pile, comme dans le cas d'un appel de fonction “classique”. La fonction, nommée `sys_getpid`, peut donc être écrite en C.

Au retour de la fonction, la valeur de retour est placée à la place de `EAX` là où les registres ont été sauvegardés sur la pile noyau. L'instruction `iret` permet de restaurer les registres et de repasser en mode utilisateur, juste après l'interruption. La fonction de la bibliothèque C peut alors retourner au programme appelant.

CAS D'ÉTUDE

On a vu que les appels systèmes permettent aux programmes utilisateur d'accéder aux services du noyau. Ils forment donc une interface particulièrement sensible aux problèmes de sécurité.

Comme pour toutes les interfaces, on peut être plus ou moins fin. D'un côté, une interface pas assez fine serait trop restrictive et ne permettrait pas d'implémenter tout type de logiciel. De l'autre, une interface trop laxiste ("écrire dans tel registre matériel") empêche toute isolation. Il faut donc trouver la bonne granularité.

Nous allons présenter ici une difficulté liée à la manipulation de mémoire au sein de certains types d'appels système.

Il y a deux grands types d'appels systèmes : d'une part, ceux qui renvoient un simple nombre, comme `getpid` qui renvoie le numéro du processus appelant.

```
pid_t pid = getpid();  
printf("%d\n", pid);
```

Ici, pas de difficulté particulière : la communication entre le *ring* 0 et le *ring* 3 est faite uniquement à travers les registres, comme décrit dans le chapitre précédent.

Mais la plupart des appels systèmes communiquent de l'information de manière indirecte, à travers un pointeur. Ainsi la fonction `gettimeofday` a pour prototype :

```
int gettimeofday(struct timeval *tv, struct timezone *tz);
```

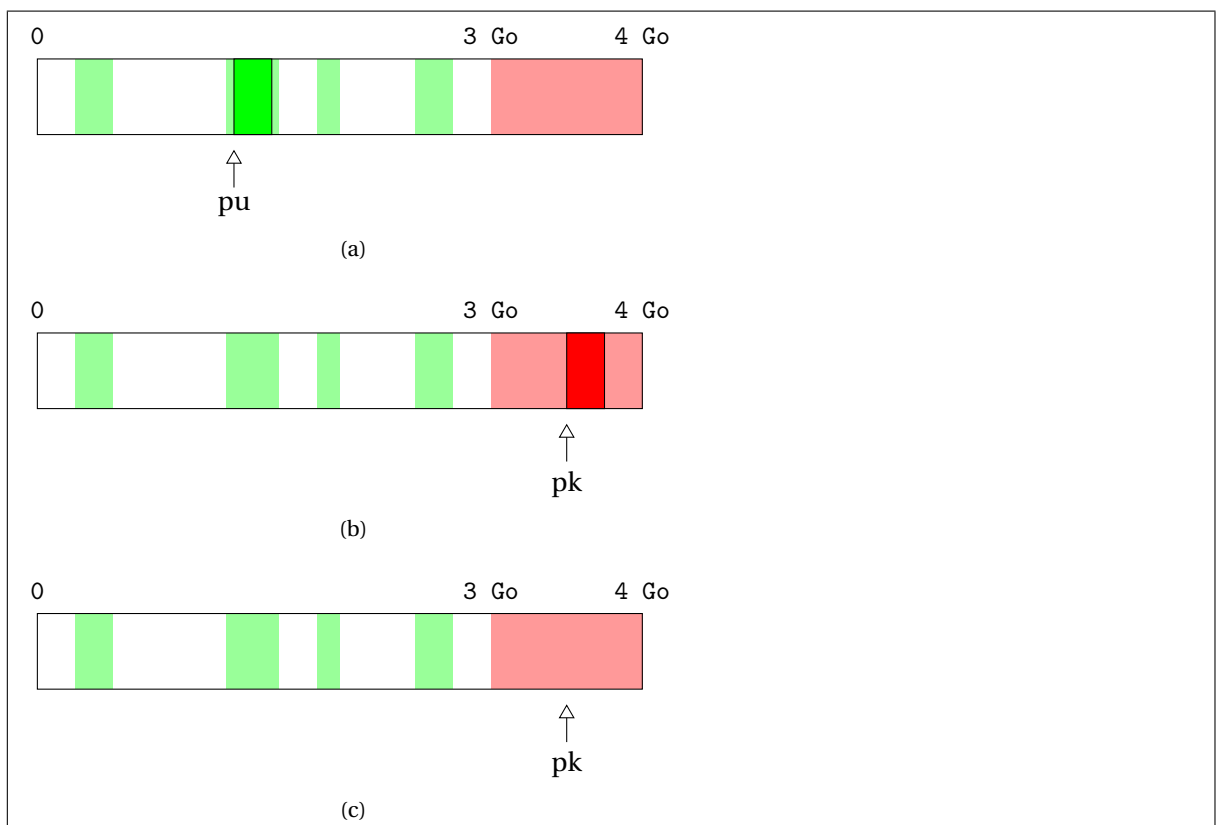
Dans le cas d'un appel système qui ne fait que renvoyer un nombre, il n'y a pas de difficulté. En revanche, certains appels système remplissent une structure avec leurs résultats. C'est alors la responsabilité de l'appelant (le programmeur utilisateur) d'allouer une zone mémoire correcte pour contenir le résultat (figure 3.1).

Puisque c'est le noyau qui implémente l'appel système, le déréférencement se déroule en *ring* 0. Il se pose alors le problème suivant : le noyau va pouvoir écrire à toute adresse fournie par l'utilisateur.

Notons que dans ce cas, c'est le noyau qui remplit la structure : le déréférencement se fait en *ring* 0.

```
struct timeval tv;
struct timezone tz;
int z = gettimeofday(&tv, &tz);
if (z == 0) {
    printf( "tv.tv_sec = %ld\ntv.tv_usec = %ld\n"
           "tz.tz_minuteswest = %d\ntz.tz_dsttime = %d\n",
           tv.tv_sec, tv.tv_usec,
           tz.tz_minuteswest, tz.tz_dsttime
    );
}
```

FIGURE 3.1 – Appel de gettimeofday



3.1 Description du problème

Un système d'exploitation moderne comme GNU/Linux est séparé en deux niveaux de privilèges : le noyau, qui gère directement le matériel, et les applications de l'utilisateur, qui communiquent avec le noyau par l'interface restreinte des *appels système*.

Pour assurer l'isolation, ces deux parties n'ont pas accès aux mêmes zones mémoire (cf. figure 2.5).

Si le code utilisateur tente d'accéder à la mémoire du noyau, une erreur sera déclenchée. En revanche, si cette écriture est faite au sein de l'implantation d'un appel système, il n'y aura pas d'erreur puisque le noyau a accès à toute la mémoire : l'isolation aura donc été brisée.

Pour celui qui implante un appel système, il faut donc empêcher qu'un pointeur passé en paramètre référence le noyau. Autrement dit, il est indispensable de vérifier dynamiquement que la zone dans laquelle pointe le paramètre est accessible par l'appelant [Har88].

Si au contraire un tel pointeur est déréférencé sans vérification (avec `*` ou une fonction comme `mempcpy`), le code s'exécutera correctement mais en rendant le système vulnérable, comme le montre la figure 3.2.

Pour éviter cela, le noyau fournit un ensemble de fonctions qui permettent de vérifier dynamiquement la valeur d'un pointeur avant de le déréférencer. Par exemple, dans la figure précédente, la ligne 8 aurait dû être remplacée par :

```
copy_from_user(&value, value_ptr, sizeof(value));
```

L'analyse présentée ici permet de vérifier automatiquement et statiquement que les pointeurs qui proviennent de l'espace utilisateur ne sont déréférencés qu'à travers une de ces fonctions sûres.

3.2 Principes de l'analyse

Le problème est modélisé de la façon suivante : on associe à chaque variable `x` un type de données `t`, ce que l'on note `x : t`. En plus des types présents dans le langage C, on ajoute une distinction supplémentaire pour les pointeurs. D'une part, les pointeurs "noyau" (de type `t *`) sont créés en prenant l'adresse d'un objet présent dans le code source. D'autre part, les pointeurs "utilisateurs" (leur type est noté `t_user*`) proviennent des interfaces avec l'espace utilisateur.

Il est sûr de déréférencer un pointeur noyau, mais pas un pointeur utilisateur. L'opérateur `*` prend donc un `t *` en entrée et produit un `t`.

Pour faire la vérification de type sur le code du programme, on a besoin de quelques règles. Tout d'abord, les types suivent le flot de données. C'est-à-dire que si on trouve dans le code `a = b`, `a` et `b` doivent avoir un type compatible. Ensuite, le qualificateur `user` est récursif : si on a un pointeur utilisateur sur une structure, tous les champs pointeurs de la structure sont également utilisateur. Enfin, le déréférencement s'applique aux pointeurs noyau seulement : si le code contient l'expression `*x`, alors il existe un type `t` tel que `x : t*` et `*x : t`.

Appliquons ces règles à l'exemple de la figure 3.2 : on suppose que l'interface avec l'espace utilisateur a été correctement annotée. Cela permet de déduire que `data : void user*`. En appliquant la première règle à la ligne 6, on en déduit que `info : struct drm_radeon_info user*` (comme en C, on peut toujours convertir de et vers un pointeur sur `void`).

Pour déduire le type de `value_ptr` dans la ligne 7, c'est la deuxième règle qu'il faut appliquer : le champ `value` de la structure est de type `uint32_t *` mais on y accède à travers un pointeur utilisateur, donc `value_ptr : uint32_t user*`.

```

/* from drivers/gpu/drm/radeon/radeon_kms.c */
int radeon_info_ioctl(struct drm_device *dev, void *data, struct drm_file *filp)
{
    struct radeon_device *rdev = dev->dev_private;
    struct drm_radeon_info *info;
    struct radeon_mode_info *minfo = &rdev->mode_info;
    uint32_t *value_ptr;
    uint32_t value;
    struct drm_crtc *crtc;
    int i, found;

    info = data;
    value_ptr = (uint32_t *)((unsigned long)info->value);
    value = *value_ptr;
    switch (info->request) {
    case RADEON_INFO_DEVICE_ID:
        value = dev->pci_device;
        break;
    case RADEON_INFO_NUM_GB_PIPES:
        value = rdev->num_gb_pipes;
        break;
    case RADEON_INFO_NUM_Z_PIPES:
        value = rdev->num_z_pipes;
        break;
    case RADEON_INFO_ACCEL_WORKING:
        /* xf86-video-ati 6.13.0 relies on this being false for evergreen */
        if ((rdev->family >= CHIP_CEDAR) && (rdev->family <= CHIP_HEMLOCK))
            value = false;
        else
            value = rdev->accel_working;
        break;
    case RADEON_INFO_CRTC_FROM_ID:
        for (i = 0, found = 0; i < rdev->num_crtc; i++) {
            crtc = (struct drm_crtc *)minfo->crtcs[i];
            if (crtc && crtc->base.id == value) {
                struct radeon_crtc *radeon_crtc = to_radeon_crtc(crtc);
                value = radeon_crtc->crtc_id;
                found = 1;
                break;
            }
        }
        if (!found) {
            DRM_DEBUG_KMS("unknown crtc id %d\n", value);
            return -EINVAL;
        }
        break;
    case RADEON_INFO_ACCEL_WORKING2:
        value = rdev->accel_working;
        break;
    case RADEON_INFO_TILING_CONFIG:
        if (rdev->family >= CHIP_CEDAR)
            value = rdev->config.evergreen.tile_config;
        else if (rdev->family >= CHIP_RV770)
            value = rdev->config.rv770.tile_config;
        else if (rdev->family >= CHIP_R600)
            value = rdev->config.r600.tile_config;
        else {

```

À la ligne 8, on peut appliquer la troisième règle : à cause du déréférencement, on en déduit que `value_ptr:t *`, ce qui est une contradiction puisque d'après les lignes précédentes, `value_ptr:uint32_t user*`.

Si la ligne 3 était remplacée par l'appel à `copy_from_user`, il n'y aurait pas d'erreur de typage car cette fonction peut accepter les arguments (`uint32_t *`, `uint32_t user*`, `size_t`).

3.3 Implantation

Une implantation est en cours. Le code source est d'abord prétraité par `gcc -E` puis converti en Newspeak [HL08], un langage destiné à l'analyse statique. Ce traducteur peut prendre en entrée tout le langage C, y compris de nombreuses extensions GNU utilisées dans le noyau. En particulier, l'exemple de la figure 3.2 peut être analysé.

À partir de cette représentation du programme et d'un ensemble d'annotations globales, on propage les types dans les sous-expressions jusqu'aux feuilles.

Si aucune contradiction n'est trouvée, c'est que le code respecte la propriété d'isolation. Sinon, cela peut signifier que le code n'est pas correct, ou bien que le système de types n'est pas assez expressif pour le code en question.

Le prototype, disponible sur <http://penjili.org>, fera l'objet d'une démonstration.

3.4 Conclusion

Nous avons montré que le problème de la manipulation de pointeurs non sûrs peut être traité avec une technique de typage. Elle est proche des analyses menées dans CQual [FFA99] ou Sparse [TTL].

Plusieurs limitations sont inhérentes à cette approche : notamment, la présence d'unions ou de *casts* entre entiers et pointeurs fait échouer l'analyse.

Le principe de cette technique (associer des types aux valeurs puis restreindre les opérations sur certains types) peut être repris. Par exemple, si on définit un type "numéro de bloc" comme étant un nouvel alias de `int`, on peut considérer que multiplier deux telles valeurs est une erreur.

ÉTAT DE L'ART

- Analyse légère :
- Sparse [TTL]
- Coccinelle
- Analyse de valeurs :
- IA :
 - Cousot 77 [CC77]
 - Cousot 92 [CC92]
- Typage fort
- [Pie02]
- Qualifiers in Haskell :
 - [KcS07]
 - [LZ06]
- Qualificateurs de type :
- CQual [FFA99, STFW01, FTA02, JW04, FJKA06]

Deuxième partie

Analyse statique par typage

5.1 Présentation et but

Au plus bas niveau d'abstraction, un ordinateur ne manipule que des nombres entiers¹ : en langage machine, il n'y a pas de distinction entre une adresse et un nombre.

Pourtant il est clair que certaines opérations n'ont pas de sens : par exemple, ajouter deux adresses, ou déréférencer le résultat d'une division sont des comportements qu'on voudrait pouvoir empêcher.

En un mot, le but du typage est de classer les objets et de restreindre les opérations possibles selon la classe d'un objet : "ne pas ajouter des pommes et des oranges".

Le modèle qui permet cette classification est appelé *système de types* et est en général constitué d'un ensemble de *règles de typage*, comme "un entier plus un entier égale un entier".

5.2 Taxonomie

La définition d'un langage de programmation introduit la plupart du temps celle d'un système de types. Il y a donc de nombreux systèmes de types différents, dont nous pouvons donner une classification sommaire.

Dynamique, statique, mixte

Il y a deux grandes familles de systèmes de types, selon quand se fait la vérification de types. On peut en effet l'effectuer au moment de l'exécution, ou au contraire prévenir les erreurs à l'exécution en la faisant au moment de la compilation (ou avant l'interprétation).

Typage dynamique

La première est le typage *dynamique*. Pour différencier les différents types de données, on ajoute une étiquette à chaque valeur. Dans tout le programme, on ne manipulera que des valeurs étiquetées :

1. en allant plus loin on pourrait dire qu'il ne manipule que des suites de bits

```
def f(b):
    x = None
    r = None
    if b:
        x = 1
    else:
        x = lambda y: y + 1
    b = not b
    if b:
        r = x (1)
    else:
        r = x + 1
    return r
```

FIGURE 5.1 – Fonction Python non typable statiquement.

- si on veut réaliser l'opération $(0x00000001, Int) + (0x0000f000, Int)$, on vérifie tout d'abord qu'on peut réaliser l'opération $+$ entre deux *Int*. Ensuite on réalise l'opération elle même, qu'on étiquette avec le type du résultat : $(0x0000f001, Int)$
- si au contraire on tente d'ajouter deux adresses $(0x2e8d5a90, Addr) + (0x76a5e0ec, Addr)$, la vérification échoue et l'opération s'arrête avec une erreur.

Il existe plusieurs techniques pour signaler les erreurs de typage dynamiques : arrêter l'exécution, lever une exception, convertir une opérande, utiliser une valeur d'erreur, etc.

Typage statique

La seconde technique est le typage *statique* : plutôt que de vérifier les types sur les données, on les vérifie à l'arrêt, sur les expressions. Cela implique par exemple que chaque variable doit contenir des valeurs d'un même type tout au long de sa vie.

A première vue, cela semble moins puissant que le typage dynamique : en effet, il existe des programmes qui s'exécuteront sans erreur de type mais sur lesquels le typage statique ne peut s'appliquer.

Dans la figure 5.1, on peut voir par une simple analyse de cas que si on fournit un booléen à *f*, elle retourne un entier. Mais selon la valeur de *b*, la variable *x* contiendra une valeur de type entier ou fonction.

Dans le cas où le typage statique est possible, les garanties sont en revanche plus importantes : les valeurs portées par une variable auront toujours le même type. Par voie de conséquence, la vérification dynamique de types réussira toujours : on peut donc la supprimer. Il est également possible de supprimer toutes les étiquettes de typage : on parle de *type erasure*.

Une conséquence heureuse de cette suppression est que l'exécution de ce programme se fera de manière plus rapide.

Connaître les types à la compilation permet aussi de réaliser plus d'optimisations. Par exemple, en Python, considérons l'expression $y = x - x$. Sans information sur le type de *x*, aucune simplification n'est possible : l'implémentation de la différence sur ce type est une fonction quelconque, sans propriétés particulières *a priori*.

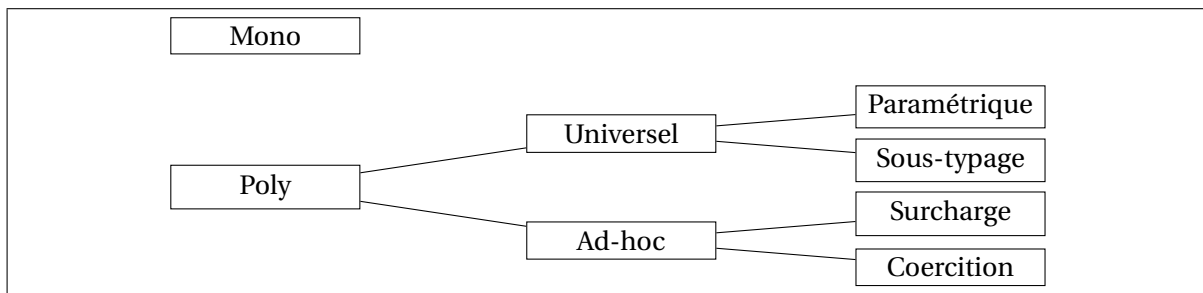


FIGURE 5.2 – Les différents types de polymorphisme.

Si au contraire, on sait que x est un entier, on peut en déduire que $y = 0$, sans réaliser la soustraction (si c'était la seule utilisation de x , le calcul de x aurait alors pu être éliminé).

Typage hybride ("stanamique")

Il est aussi possible de mélanger ces deux approches :

- à la compilation, essayer d'obtenir les types les plus précis possibles
- pour les cas restants, insérer un test dynamique

Fort, faible, sound

Un autre critère intéressant est la "force" du typage : est-ce qu'il peut induire des conversions de type implicites ?

Prenons l'exemple de l'addition entre un entier 2 et un flottant 3.14. On peut le voir de plusieurs manières différentes :

- un entier est un nombre décimal comme les autres : $2 = 2.00$. Donc $2 + 3.14 = 2.00 + 3.14 = 5.14$.
- 2 est un entier, 3.14 un flottant : ces deux types sont différents et l'opération est impossible.

Polymorphisme

Dans le cas du typage statique, restreindre une opération à un seul type de données peut être assez restrictif.

Par exemple, quel doit être le type d'une fonction qui trie un tableau ?

Monomorphisme

Une première solution peut être de forcer des types concrets, c'est à dire qu'une même fonction ne pourra s'appliquer qu'à un seul type de données.

Il est confortable pour le programmeur de n'avoir à écrire un algorithme qu'une seule fois, indépendamment du type des éléments considérés.

Il existe deux grandes classes de systèmes de types introduisant du polymorphisme.

Polymorphisme universel

Le polymorphisme est dit universel si toute fonction générique peut s'appliquer à n'importe quel type.

```

let rec append lx ly =
  match lx with
  | [] -> ly
  | x::xs -> x::append lx ly

```

FIGURE 5.3 – Fonction de concaténation de listes en OCaml.

Polymorphisme ad-hoc

Le polymorphisme est *ad-hoc* si les fonctions génériques ne peuvent s'appliquer qu'à un ensemble de types prédéfini.

Polymorphisme paramétrique

Historique + citer le papier de Milner sur le polymorphisme

La fonction de la figure 5.3 n'opère que sur la structure du type liste (en utilisant ses constructeurs [] et (: :)) ainsi que le filtrage) : les éléments de lx et ly ne sont pas manipulés à part pour les transférer dans le résultat.

Moralement, cette fonction est donc indépendante du type de données contenu dans la liste : elle pourra agir sur des listes de n'importe quel type d'élément.

Plutôt qu'un type, on peut lui donner le *schéma de types* suivant :

$$\text{append} : \forall a. \text{alist} \rightarrow \text{alist} \rightarrow \text{alist}$$

C'est à dire que append peut être utilisé avec n'importe quel type concret a en substituant les variables quantifiées (on parle d' *instanciation*).

Polymorphisme par sous-typage

héritage, sous-typage, classe, méthode multiple, late binding, Liskov

■ Dans les langages définissent la notion de sous-typage. C'est une relation d'ordre partiel sur les types, qui modélise la relation "est un". Chaque sous-classe peut redéfinir le comportement de chaque méthode de ses superclasses.

Polymorphisme par surcharge

Considérons l'opération d'addition : +. On peut considérer que certains types l'implémentent, et pas d'autres : ajouter deux flottants ou deux entiers a du sens, mais pas ajouter deux pointeurs.

On dira que + est *surchargé*. À chaque site d'appel, il faudra *résoudre la surcharge* pour déterminer quelle fonction appeler.

Introduire l'inférence plus haut

■ Cela rend l'inférence de types impossible dans le cas général, puisque certaines constructions sont ambiguës.

Dans le code Haskell de la figure 5.4, show peut s'appliquer à toutes les valeurs de types "affichables" et renvoie une représentation textuelle. read réalise le contraire avec les types "lisibles".

Lorsqu'on compose ces deux fonctions, le type de la valeur intermédiaire est capital puisqu'il détermine les instances de show et read à utiliser.

```
show :: Show a => a -> String
read :: Read a => String -> a

showRead :: String -> String
showRead x = show (read x)
```

FIGURE 5.4 – Cas d'ambiguïté avec de la surcharge ad-hoc.

Polymorphisme par coercition**Polymorphisme d'ordre supérieur**

`g f = (f true, f 2)`

$$g : (\forall a. a \rightarrow a) \rightarrow (bool * int)$$

Pas inférable (annotations nécessaires).

Expressivité, garanties, types dépendants**5.3 Exemples**

Faible dynamique : Perl

Faible statique : C

Fort dynamique : Python

Fort statique : OCaml

Fort statique à effets typés : Haskell

Theorem prover : Coq

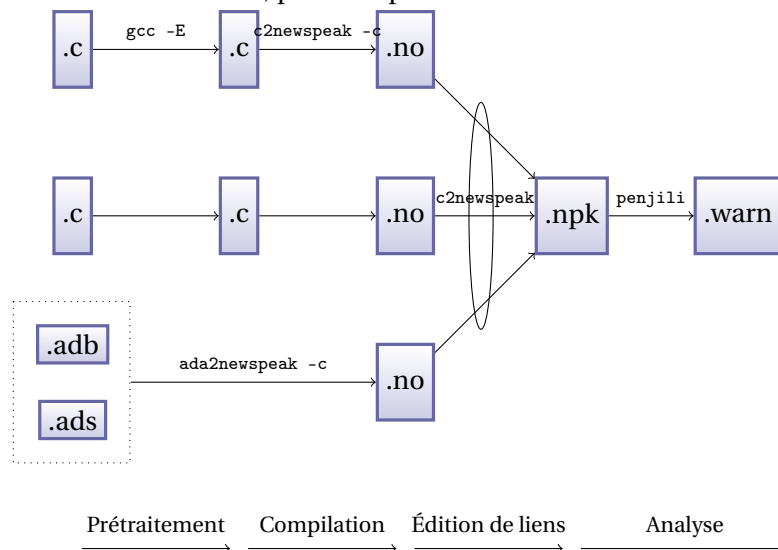
UN PREMIER SYSTÈME DE TYPES

6.1 But

$\vdash \text{memcpy} : \forall a.(a^*, a^*, \text{size_t}) \rightarrow ()$

6.2 Chaîne de compilation

La compilation vers C est faite en trois étapes : prétraitement du code source, compilation de C prétraité vers NEWSPEAK, puis compilation de NEWSPEAK vers ce langage.



Prétraitement

C2NEWSPEAK travaillant uniquement sur du code prétraité (dans directives de préprocesseur), la première étape consiste donc à faire passer le code par CPP.

Compilation (levée des ambiguïtés)

Cette passe est réalisée par l'utilitaire C2NEWSPEAK. L'essentiel de la compilation consiste à mettre à plat les définitions de types, et à simplifier le flot de contrôle. C en effet propose de nombreuses constructions ambiguës ou redondantes.

Au contraire, NEWSPEAK propose un nombre réduit de constructions. Rappelons que le but de ce langage est de faciliter l'analyse statique : des constructions orthogonales permettent donc d'éviter la duplication de règles sémantiques, ou de code lors de l'implémentation d'un analyseur.

Par exemple, plutôt que de fournir une boucle `while`, une boucle `do/while` et une boucle `for`, NEWSPEAK fournit une unique boucle `WHILE(1){}`. La sortie de boucle est compilée vers un `GOTO`, qui est toujours un saut vers l'avant (similaire à un "break" généralisé).

La sémantique de NEWSPEAK et la traduction de C vers NEWSPEAK sont décrites dans [HL08]. En ce qui concerne l'élimination des sauts vers l'arrière, on peut se référer à [EH94].

Annotations

NEWSPEAK a de nombreux avantages, mais pour une analyse par typage il est trop bas niveau. Par exemple, dans le code suivant

```
struct s {
    int a;
    int b;
};

int main(void)
{
    struct s x;
    int y[10];
    x.b = 1;
    y[1] = 1;
    return 0;
}
```

6.3 Syntaxe

La grammaire suivante définit un langage impératif. On suppose qu'on peut compiler un programme écrit en C vers ce langage.

Un programme est un triplet $P = (\vec{f}, \vec{x}, b)$ constitué d'un ensemble de fonctions, d'un ensemble de variables et d'un bloc d'instructions. Ce bloc sera exécuté au lancement du programme ; il peut par exemple contenir le code d'initialisation des variables globales et l'appel à la fonction principale.

$\langle \text{programme} \rangle ::= (\text{fonctions}, \text{globales}, \langle \text{bloc} \rangle)$

$\langle expr \rangle ::= \langle lval \rangle$	$\langle bloc \rangle ::= \langle instr \rangle ; \langle bloc \rangle$
unop $\langle expr \rangle$	ϵ
$\langle expr \rangle$ binop $\langle expr \rangle$	$\langle instr \rangle ::= \langle lval \rangle \leftarrow \langle expr \rangle$
cst	$\langle lval \rangle \leftarrow \text{funexp}(\text{args})$
$\& \langle lval \rangle$	funexp (args)
$\& \text{fonction}$	$\uparrow \text{nom} \{ \langle bloc \rangle \}$
$\langle lval \rangle ::= \text{var}$	if ($\langle expr \rangle$) { $\langle bloc \rangle$ } else { $\langle bloc \rangle$ }
$\langle lval \rangle . \text{champ}$	{ $\langle bloc \rangle$ } label:
$\langle lval \rangle [\langle expr \rangle]$	goto label
$* \langle expr \rangle$	forever { $\langle bloc \rangle$ }
	return $\langle expr \rangle$

Contrairement au langage C, le langage des expressions et des instructions est séparé. Par conséquence, l'évaluation des expressions peut se faire sans effet de bord.

expliquer pourquoi un while expr ne suffit pas

6.4 Sémantique (opérationnelle, à petits pas)

On définit une sémantique opérationnelle à petits pas, sous forme d'une relation de transition (notée \rightarrow) entre états de l'interpréteur.

Ces états sont constitués des composantes suivantes :

- un point de contrôle l dans le programme. Ils sont issus d'une première transformation en un graphe de flot de contrôle.
- un état mémoire σ qui associe à chaque variable une valeur.

Graphe de flot de contrôle

Dans la syntaxe ci-dessus, on peut classer les instructions en deux familles : celles qui définissent le flot de contrôle (if, dowith, goto, forever) et celles qui définissent le flot de données. Une première transformation va transformer chaque fonction en son graphe de flot de contrôle, défini comme suit :

- les nœuds sont des points de contrôle, qui représentent par exemple l'adresse mémoire de l'instruction qui vient d'être exécutée.
- les arêtes sont soit des instructions "de données" (affectation, appel de fonction, déclaration), soit des conditions (ie une expression).

```

int gcd(int a, int b)
{
    if (a == 0) {
        return b;
    }
    while (b != 0) {
        if (a > b) {
            a = a - b;
        } else {
            b = b - a;
        }
    }
}

xxx

int32 gcd(int32 a, int32 b) {
    if (a == 0) {
        !return = b_int32;
        goto lbl0;
    }
    while (1) {
        if (b == 0) {
            goto lbl1;
        }
        if (a > b) {
            a = a - b;
        } else {
            b = b - a;
        }
    }
    lbl1:
    lbl0:
    ;
}

```

Intuitivement, on peut "passer" d'un état à un autre soit en passant par une arête "condition" qui s'évalue à une valeur "vrai", soit en appliquant les effets de bord d'une arête "instruction".

Dans la suite, on suppose qu'on a à notre disposition un ensemble de jugements : $\langle l, instr, l' \rangle$ qui signifie qu'on peut passer du point l au point l' en effectuant l'instruction $instr$.

État mémoire

La mémoire interne de l'interpréteur est une correspondance entre l'ensemble des adresses (infini dénombrable) et l'ensemble des valeurs. Un état mémoire σ est une fonction partielle de $Addr$ vers Val .

Ces valeurs peuvent être de plusieurs formes :

$v ::= n, f, \text{NIL}$	constantes
$\&a$	pointeur vers l'adresse a
$\&f$	pointeur vers la fonction f
\top	valeur non initialisée

file d'appels

Left values

La mémoire est organisée en adresses, mais pourtant dans le programme cette notion n'est pas directement visible. Les accès sont réalisés à travers des "left values".

Jugements

Les jugements ont les formes suivantes :

- $\sigma \vdash lv \Rightarrow a$: la left-value lv correspond à l'adresse mémoire a .
- $\sigma \vdash e \Rightarrow v$: l'expression e s'évalue en v .
- $(l, \sigma) \rightarrow (l', \sigma')$: permet de définir la fonction de transition principale

Sémantique des left-values

$$\frac{(v, a) \in \sigma}{\sigma \vdash v \Rightarrow a} \text{ (EVAL-LV-VAR)} \quad \frac{\sigma \vdash e \Rightarrow \&a}{\sigma \vdash *e \Rightarrow a} \text{ (EVAL-LV-DEREF)}$$

$$\frac{\sigma \vdash lv \Rightarrow a}{\sigma \vdash lv.f \Rightarrow a + f} \text{ (EVAL-LV-FIELD)} \quad \frac{\sigma \vdash lv \Rightarrow a \quad \sigma \vdash e \Rightarrow n}{\sigma \vdash lv[e] \Rightarrow a + n} \text{ (EVAL-LV-ARRAY)}$$

Sémantique des expressions

$$\frac{}{\sigma \vdash c \Rightarrow c} \text{ (EVAL-CST)} \quad \frac{\sigma \vdash lv \Rightarrow a \quad (a, v) \in \sigma}{\sigma \vdash lv \Rightarrow v} \text{ (EVAL-LV)}$$

$$\frac{\sigma \vdash e \Rightarrow v}{\sigma \vdash ope \Rightarrow \widehat{op}v} \text{ (EVAL-UNOP)} \quad \frac{\sigma \vdash e_1 \Rightarrow v_1 \quad \sigma \vdash e_2 \Rightarrow v_2}{\sigma \vdash e_1 ope_2 \Rightarrow v_1 \widehat{op} v_2} \text{ (EVAL-BINOP)}$$

$$\frac{\sigma \vdash lv \Rightarrow a}{\sigma \vdash \&lv \Rightarrow \&a} \text{ (EVAL-ADDR OF)} \quad \frac{}{\sigma \vdash \&f \Rightarrow \&f} \text{ (EVAL-ADDR OF FUN)}$$

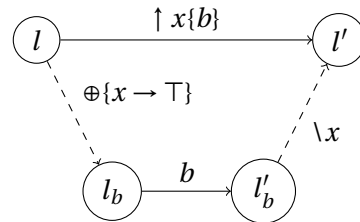
Sémantique des instructions

La règle la plus simple concerne l'affectation : on peut affecter une expressions à une left value si elles ont le même type.

$$\frac{\langle l, lv \leftarrow e, l' \rangle \quad \sigma \vdash lv \Rightarrow a \quad \sigma \vdash e \Rightarrow v}{(l, \sigma) \rightarrow (l', \sigma[a \mapsto v])} \text{ (INSTR-ASSIGN)}$$

Déclarer une variable, c'est rendre accessible dans un bloc une variable non initialisée, qui n'est plus accessible par la suite : Si on suppose qu'on peut traverser le bloc interne b sous un σ enrichi d'une nouvelle variable x , on peut donc traverser l'instruction $\uparrow x\{b\}$.

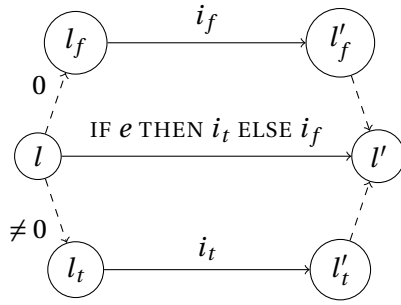
$$\frac{\langle l, \uparrow x\{b\}, l' \rangle \quad \langle l_b, b, l'_b \rangle \quad \sigma' = \sigma \oplus \{x \mapsto \top\} \quad (l_b, \sigma', s) \rightarrow (l'_b, \sigma'')}{(l, \sigma) \rightarrow (l', \sigma'' \setminus x)} \text{ (INSTR-DECL)}$$



fcall

Sémantique des conditions

On utilise un encodage similaire à la déclaration. Tout d'abord, on évalue la condition dans un contexte σ . Si elle s'évalue en un entier non nul, et qu'une transition à travers le bloc i_t est possible, alors on peut faire passer à travers le "if".



$$\frac{\langle l, \text{IF } e \text{ THEN } i_t \text{ ELSE } i_f, l' \rangle \quad \sigma \vdash e \Rightarrow n \quad n \neq 0 \quad \langle l_i, i_t, l'_i \rangle \quad (l_i, \sigma) \rightarrow (l'_i, \sigma')}{(l, \sigma) \rightarrow (l', \sigma')} \text{ (IF-TRUE)}$$

$$\frac{\langle l, \text{IF } e \text{ THEN } i_t \text{ ELSE } i_f, l' \rangle \quad \sigma \vdash e \Rightarrow 0 \quad \langle l_i, i_f, l'_i \rangle \quad (l_i, \sigma) \rightarrow (l'_i, \sigma')}{(l, \sigma) \rightarrow (l', \sigma')} \text{ (IF-FALSE)}$$

6.5 Règles de typage

Dans cette section, on définit la notion de programme bien typé. L'analyse par typage permet de vérifier qu'à chaque expression on peut associer un type, et ce de manière cohérente entre plusieurs utilisations d'une variable.

Un jugement de typage est de la forme $\Gamma \vdash e : \tau$ et se lit "sous Γ , e est typable en τ ". Un environnement de typage Γ contient le contexte nécessaire à l'analyse, c'est à dire le type des fonctions et variables du programme.

Les instructions et blocs, au contraire, n'ont pas de type. On note $\Gamma \vdash i$ pour "sous Γ , i est bien typé", c'est à dire que ces sous expressions sont typables et que les types sont en accord avec le flot de données (par exemple, pour que l'instruction $lv \leftarrow e$ soit bien typée sous Γ , il faut que les types de lv et de e puissent avoir le même type sous Γ).

Les types des valeurs sont :

$\tau ::= \text{INT, FLOAT, VOID}$	constantes
$ a$	variable
$ (\tau_1, \dots, \tau_n) \rightarrow \tau_r$	fonction
$ [\tau]$	tableau
$ \tau^*$	pointeur
$ \{f_1 : \tau_1, \dots, f_n : \tau_n\}$	structure

Schémas de type

On va associer à chaque variable globale un type. Mais faire de même pourrait être trop restrictif. En effet, une fonction comme `memcpy` peut être utilisée pour copier des tableaux d'entiers, mais aussi de flottants. On va donc associer un schéma de types à chaque fonction.

$$\sigma ::= \forall \vec{a}. \tau$$

En associant un schéma de type σ à une fonction f , on indique que la fonction pourra être utilisée avec tout type τ qui est une instantiation de σ .

Programme

Au niveau global, un programme P est bien typé (noté $\vdash P$) s'il existe un environnement Γ^0 permettant de typer ses composantes (les fonctions, les globales et le bloc d'initialisation).

$$\frac{\Gamma^0 = (\vec{\sigma}, \vec{\tau}) \quad \Gamma^0 \vdash \vec{f} : \vec{\sigma} \quad \Gamma^0 \vdash \vec{x} : \vec{\tau} \quad \Gamma^0 \vdash b}{\vdash (\vec{f}, \vec{x}, b)} \text{ (PROG)}$$

Flot de contrôle

Les règles suivantes permettent de définir les jugements $\Gamma \vdash i$. En résumé, les instructions sont bien typées si leurs sous-instructions sont bien typées. La seule règle supplémentaire concerne la condition du if qui doit être typée en INT.

$$\begin{array}{c} \frac{}{\Gamma \vdash \epsilon} \text{ (PASS)} \quad \frac{\Gamma \vdash s \quad \Gamma \vdash b}{\Gamma \vdash s; b} \text{ (SEQ)} \\ \frac{\Gamma \vdash b}{\Gamma \vdash \text{forever}\{b\}} \text{ (FOREVER)} \quad \frac{\Gamma \vdash e : \text{INT} \quad \Gamma \vdash i_t \quad \Gamma \vdash i_f}{\Gamma \vdash \text{IF } e \text{ THEN } i_t \text{ ELSE } i_f} \text{ (IF)} \\ \frac{}{\Gamma \vdash \text{gotol}} \text{ (GOTO)} \quad \frac{\Gamma \vdash b}{\Gamma \vdash \{b\}l :} \text{ (DOWITH)} \end{array}$$

Left values

On étend la relation de typage aux left values : chaque left value, vue comme une expression, peut être typée.

$$\begin{array}{c} \frac{(v, \tau) \in \Gamma}{\Gamma \vdash v : \tau} \text{ (LV-VAR)} \quad \frac{\Gamma \vdash lv : \tau_s \quad (f, \tau_f) \in \tau_s}{\Gamma \vdash lv.f : \tau_f} \text{ (LV-FIELD)} \\ \frac{\Gamma \vdash e : \tau^*}{\Gamma \vdash *e : \tau} \text{ (LV-DEREF)} \quad \frac{\Gamma \vdash lv : [\tau] \quad \Gamma \vdash e : \text{INT}}{\Gamma \vdash lv[e] : \tau} \text{ (LV-ARRAY)} \end{array}$$

Expressions

$$\begin{array}{c}
\frac{}{\Gamma \vdash n : \text{INT}} \text{ (CONST-INT)} \\
\frac{}{\Gamma \vdash f : \text{FLOAT}} \text{ (CONST-FLOAT)} \\
\frac{}{\Gamma \vdash \text{NIL} : \tau^*} \text{ (CONST-NIL)} \\
\frac{op \in \{+, -, \ddot{O}, /, \&, |, \&\&, ||, \lll, \ggg\} \quad \Gamma \vdash e_1 : \text{INT} \quad \Gamma \vdash e_2 : \text{INT}}{\Gamma \vdash e_1 op e_2 : \text{INT}} \text{ (OP-INT)} \\
\frac{op \in \{+., -., \ddot{O}., /., \} \quad \Gamma \vdash e_1 : \text{FLOAT} \quad \Gamma \vdash e_2 : \text{FLOAT}}{\Gamma \vdash e_1 op e_2 : \text{FLOAT}} \text{ (OP-FLOAT)} \\
\frac{op \in \{=, \neq, \leq, \geq, <, >\} \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau \quad \tau \in \{\text{INT}, \text{FLOAT}\}}{\Gamma \vdash e_1 op e_2 : \text{INT}} \text{ (OP-CMP)} \\
\frac{\tau \in \{\text{INT}, \text{FLOAT}\} \quad \Gamma \vdash e : \tau}{\Gamma \vdash -e : \tau} \text{ (UNOP-MINUS)} \\
\frac{op \in \{\!, \!\} \Gamma \vdash e : \text{INT}}{\Gamma \vdash op e : \text{INT}} \text{ (UNOP-NOT)} \\
\frac{\Gamma \vdash lv : \tau}{\Gamma \vdash \&lv : \tau^*} \text{ (ADDR OF)} \\
\frac{\Gamma \vdash f : \sigma \quad \tau \leq \sigma}{\Gamma \vdash \&lv : \tau} \text{ (ADDR OF FUN)}
\end{array}$$

Fonctions

Pour typer une fonction, on commence par ajouter ses paramètres dans l'environnement de typage, et on type la définition de la fonction. Le type résultant est généralisé.

$$\frac{\Gamma' = \Gamma \oplus \{args(f) = \vec{\tau}\} \quad \Gamma' \vdash body(f) \quad \Gamma' \vdash !ret : \tau_r}{\Gamma \vdash f : Gen(\vec{\tau} \rightarrow \tau_r, \Gamma)} \text{ (FUN)}$$

Instructions

$$\begin{array}{c}
\frac{\Gamma \oplus \{x : \tau\} \vdash b}{\Gamma \vdash \uparrow x \{b\}} \text{ (DECL)} \\
\frac{\Gamma \vdash lv : \tau \quad \Gamma \vdash e : \tau}{\Gamma \vdash lv \leftarrow e} \text{ (ASSIGN)} \\
\frac{\Gamma \vdash lv : \tau_{ret} \quad \Gamma \vdash fe : \sigma \quad \Gamma \vdash \vec{e} : \vec{\tau} \quad (\vec{\tau} \rightarrow \tau_r) \leq \sigma}{\Gamma \vdash lv \leftarrow fe(\vec{e})} \text{ (FCALL)}
\end{array}$$

6.6 Limitations

Programmes non typables

Incohérences

SYSTÈME AVEC LE QUALIFICATEUR "USER"

7.1 Éditions et ajouts

7.2 Passage sur le cas d'étude

7.3 Propriété d'isolation mémoire

Le déréférencement d'un pointeur dont la valeur est contrôlée par l'utilisateur ne peut se faire qu'à travers une fonction qui vérifie la sûreté de celui-ci.

SYSTÈME AVEC LE QUALIFICATEUR "SZ"

8.1 But

8.2 Annotation de `string.h`

IMPLANTATION

CONCLUSION

- Limitations
- Perspectives

LISTE DES FIGURES

TABLE DES FIGURES

2.1	Cadres de pile	12
2.2	Les différents <i>rings</i>	13
2.3	Implémentation de la mémoire virtuelle	13
2.4	Mécanisme de mémoire virtuelle.	14
2.5	Espace d'adressage d'un processus	14
3.1	Appel de <code>gettimeofday</code>	18
3.2	Bug <code>freedesktop.org #29340</code>	20
5.1	Fonction Python non typable statiquement.	28
5.2	Les différents types de polymorphisme.	29
5.3	Fonction de concaténation de listes en OCaml.	30
5.4	Cas d'ambigüité avec de la surcharge ad-hoc.	31

BIBLIOGRAPHIE

BIBLIOGRAPHIE

- [BP05] Daniel P. Bovet and Marco Cesati Ph. *Understanding the Linux Kernel, Third Edition*. O'Reilly Media, third edition edition, November 2005.
- [CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77 : Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of Programming Languages*, pages 238–252, New York, NY, USA, 1977. ACM.
- [CC92] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(2–3) :103–179, 1992. (The editor of Journal of Logic Programming has mistakenly published the unreadable galley proof. For a correct version of this paper, see <http://www.di.ens.fr/~cousot.>).
- [EH94] Ana Erosa and Laurie J. Hendren. Taming control flow : A structured approach to eliminating goto statements. In *In Proceedings of 1994 IEEE International Conference on Computer Languages*, pages 229–240. IEEE Computer Society Press, 1994.
- [FFA99] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. A theory of type qualifiers. In *Programming language design and implementation, PLDI '99*, pages 192–203, 1999.
- [FJKA06] Jeffrey S. Foster, Robert Johnson, John Kodumal, and Alex Aiken. Flow-insensitive type qualifiers. *ACM Trans. Program. Lang. Syst.*, 28 :1035–1087, November 2006.
- [FTA02] Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-sensitive type qualifiers. In *PLDI '02 : Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, volume 37, pages 1–12, New York, NY, USA, May 2002. ACM Press.
- [Har88] Norm Hardy. The confused deputy (or why capabilities might have been invented). *ACM Operating Systems Review*, 22(4) :36–38, October 1988.
- [HL08] Charles Hymans and Olivier Levillain. Newspeak, Doubleplussimple Minilang for Good-thinkful Static Analysis of C. Technical Note 2008-IW-SE-00010-1, EADS IW/SE, 2008.
- [Int] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*.
- [JW04] Robert Johnson and David Wagner. Finding user/kernel pointer bugs with type inference. In *USENIX Security Symposium*, pages 119–134, 2004.
- [KcS07] Oleg Kiselyov and Chung chieh Shan. Lightweight static capabilities. *Electr. Notes Theor. Comput. Sci.*, 174(7) :79–104, 2007.
- [LZ06] Peng Li and Steve Zdancewic. Encoding information flow in Haskell. In *Proceedings of the 19th IEEE Workshop on Computer Security Foundations (CSFW '06)*, Washington, DC, USA, 2006. IEEE Computer Society.
- [Pie02] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

- [STFW01] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting format string vulnerabilities with type qualifiers. In *SSYM'01 : Proceedings of the 10th conference on USENIX Security Symposium*, page 16, Berkeley, CA, USA, 2001. USENIX Association.
- [Tan07] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2007.
- [TTL] Linus Torvalds, Josh Triplett, and Christopher Li. Sparse - a semantic parser for C. https://sparse.wiki.kernel.org/index.php/Main_Page.