

Méthodes et Outils pour la Conception Avancée

Castel Antonin, Reboul Paul, Vandendorpe Thomas

April 13, 2018

Contents

1	Introduction	2
2	Modularité, maintenabilité, réutilisabilité	3
2.1	Structuration	3
2.2	Documentation	4
3	Qualité des tests	5
3.1	Les tests unitaires	5
3.2	Couverture	6
4	Détection de défauts et erreurs	8
4.1	Débogueur	8
4.2	Exécution symbolique	9
5	Profiling	10
6	Analyse de performance	12
7	Analyse de vulnérabilité	14
7.1	Outil pour l'analyse de vulnérabilité	14
7.2	Exemple d'analyse de vulnérabilité	14
8	Conclusion	17

Introduction

Ce dossier rassemble les travaux effectués lors du cours de “Méthodes et Outils pour la Conception Avancée”. L’objectif ici, est de montrer les bonnes pratiques à avoir lors de la conception d’un programme. Nous utiliserons ces bonnes pratiques sur un exemple de programme de puissance 4 (en langage C), initialement codé de façon peu rigoureuse, que nous améliorerons tout au long de ce dossier. Par améliorer, on entend la maintenabilité, la réutilisabilité et la documentation du code, la qualité et la couverture des tests, la détection de défauts et d’erreurs, l’analyse et l’amélioration des performances, ainsi que l’analyse de la vulnérabilité à certaines attaques. Nous présenterons également des résultats produits par divers outils, permettant de se rendre compte des modifications apportées. Le but ici n’est pas d’expliquer le fonctionnement en détail de ces outils mais d’expliquer les différents concepts et méthodes permettant ces améliorations, et de les illustrer.

Modularité, maintenabilité, réutilisabilité

Structuration

L'élément le plus important pour gagner du temps dans notre travail, est l'organisation. La première chose que l'on commence à faire, est de structurer le code en parties pour s'y retrouver le mieux possible. Cela implique de rassembler les fonctions par thèmes dans différents fichiers. Lors de la séparation en plusieurs fichiers, les variables globales initiales sont devenues obsolètes, laissant place à un type struct plus facile d'utilisation entre les fichiers. L'application est désormais décomposée en plusieurs modules ce qui permet de faire des tests unitaires indépendamment des autres, mais aussi de trouver et corriger des erreurs plus facilement lors de la compilation.

Exemple Dans le cas de notre programme de puissance 4, nous avons séparé l'application en 5 fichiers selon les fonctions initialement présentes :

- appli.c : contient la fonction main
- board.c/board.h : les fonctions manipulant la structure board (le plateau de puissance 4)
- action.c/action.h : les actions (les règles) du jeu
- ia.c/ia.h : les IAs
- score.c/score.h : les informations de scores

Pour que l'environnement de travail soit propre et accessible, on regroupe les fichiers sources dans un même dossier, de même pour les includes, les tests et documents de sorte à faciliter la recherche des fichiers.

Documentation

Pour que le code d'une application soit compris par un large public, sans devoir expliquer chaque fonction du code en détail, on emploie une méthode de norme globale. Comme par exemple, le renommage des variables et des fonctions selon leur usage, ce qui permet une compréhension plus aisée du code. D'autre part, Doxygen permet de faire une documentation succincte, il suffit d'écrire un commentaire pour chaque fonction, décrivant son utilisation et fonctionnement. Doxygen permet d'avoir une documentation élémentaire et simple à mettre à jour.

Exemple Voici une utilisation de Doxygen, nous avons commenté notre code de la façon suivante:

```
/**
 * \file   board.c
 * \brief  Gestion de la grille de jeu ainsi que des scores
 */
/**
 * \brief   Cree une table de jeu
 * \details initialisation de la grille de jeu ainsi que des scores
 * \param   width      largeur de la grille
 * \param   heigh      longueur de la grille
 * \param   highscores  table des highscores
 * \return  une grille de jeu selon les parametres donnees
 */
Board board_new(int width,int height,int highScores){
    Board b;
    int i,j;
    b.tab = (char **)malloc(height*sizeof(char *));
    b.width = width;
    b.height = height;
    b.highScores = highScores;
    for(i=0; i<height; i++)
    {
        b.tab[i] = (char*) malloc(width*sizeof(char));
        for(j=0; j<width; j++)
        {
            b.tab[i][j]= VIDE;
        }
    }
    b.last = malloc(width*height*sizeof(char));
    b.n_last = 0;
    b.undoRedoIndex = 0;
    return b;
}
```

Après avoir généré la documentation html, on obtient:



Qualité des tests

On ne peut pas parler de qualité du logiciel sans parler des tests. Bien que ne prouvant pas l'absence de bug dans un code, les tests exécutés sur un programme sont de bons indicateurs sur sa qualité. Il est nécessaire de “penser tests” tôt dans la création d’un logiciel, car les tests sont écrits au fur et à mesure de la conception et sont destinés à être exécutés plusieurs fois, ceci afin de vérifier que les modifications et ajouts dans le code permettent toujours de valider les tests précédents. Et ceci est d’autant plus vrai lors d’un projet en équipe où les modifications des uns peuvent affecter les modifications des autres. Nous allons dans cette partie présenter deux méthodes permettant d’assurer une certaine rigueur dans les tests : l’une au sujet des tests unitaires et la seconde concernant l’analyse de couverture.

Les tests unitaires

Les tests unitaires sont des tests s’appliquant à une partie d’un programme et permettant de vérifier le programme par bloc. Un bloc de programme est en fait une fonction, et l’idée est d’écrire un test qui appelle une certaine fonction avec certains paramètres (judicieusement choisis si possible, c’est-à-dire couvrant le plus de branches possibles) et de vérifier que le résultat renvoyé par la fonction est bien le résultat attendu. Il est possible d’écrire ces tests en suivant un processus de développement dirigé par les tests, qui consiste à écrire les tests avant d’écrire la fonction correspondante. Mais on peut également les écrire si on dispose déjà de la fonction à tester en utilisant sa spécification (qui devrait être facilement accessible si le code a été documenté avec un outil adéquat), ce qui arrive si l’on est amené à retoucher un code existant.

TODO : Nécessiter de prendre en compte/modifier des fonctions pour permettre le test unitaire (valeur retour, etc.) + découpage code en fonction.

Exemple Illustrons ce principe de tests unitaires à l’aide d’un exemple tiré d’un programme de puissance 4 (en langage C). Nous souhaitons écrire un test pour la fonction suivante :

```
int checkfull(Board board);
```

```

/*
* Verifie si une grille(Board) est pleine
*
* Parameters
*   board: une table de jeu
*
* Returns
*   0 si la grille est pleine, 1 sinon
*/

```

Il existe de nombreux frameworks permettant l'automatisation des tests unitaires, dans cet exemple, on utilise CuTest. Il peut être intéressant de remarquer (en gras) l'assignation des "variable actual" (valeur renvoyée par la fonction testée), de "expected" (valeur attendue) et leur comparaison (qui fonctionne un peu comme une assertion).

```

void Test_checkfull(CuTest *tc){
    int i,j,expected,actual;
    Board b=boardVide(15,22);
    for(i=0;i<b.height;i++){
        for(j=1;j<=b.width;j++){
            actual = checkfull(b);
            expected = 1;
            CuAssertIntEquals(tc, expected, actual);
            board_put(&b,j);
        }
    }
    actual = checkfull(b);
    expected = 0;
    CuAssertIntEquals(tc, expected, actual);
}

```

Les tests écrits de cette façon pourront ensuite tous être exécutés en même temps et des informations sur les échecs des tests seront disponibles :

```

.....F.....

There was 1 failure:
1) Test_checkfull: src/CuTestTest.c:221: expected <0> but was <1>

!!!FAILURES!!!
Runs: 17 Passes: 16 Fails: 1

.....

OK (17 tests)

```






Comme les tests sont ensuite faciles à relancer, il est important de les réexécuter à chaque changement important dans le code afin d'éviter la régression.

Couverture

La couverture est un indicateur permettant de savoir quelles parties du code ont été couvertes par les tests. Utiliser la couverture seule n'est pas forcément intéressant, puisqu'elle n'indique pas si les tests ont été validés ou non. En revanche, en combinaison des tests unitaires vu précédemment, on a un indicateur

de la qualité des tests très puissant puisqu'on a l'information du pourcentage du code testé et du nombre de tests validés.

Exemple Nous avons utilisé l'outil gcov afin d'observer la couverture de notre puissance 4. Comme ce dernier ne donne qu'un résultat textuel de la couverture, nous avons complété son utilisation avec l'outil lcov, qui utilise ce que gcov a produit mais rajoute une interface html afin que la couverture soit plus facile à lire. On peut ainsi observer le pourcentage de lignes/fonctions visitées par les tests :

Filename	Line Coverage ↕	Functions ↕
actions.c	 100.0 % 39 / 39	100.0 % 2 / 2
appli.c	 93.8 % 91 / 97	100.0 % 1 / 1
board.c	 90.4 % 206 / 228	100.0 % 15 / 15
ia.c	 93.9 % 77 / 82	100.0 % 4 / 4
score.c	 100.0 % 45 / 45	100.0 % 5 / 5

En plus des pourcentages de couverture, on peut également directement voir sur le code ce qui a été couvert (en bleu) et ce qui ne l'a pas été (en rouge). Cela est pratique lorsqu'on écrit des tests en essayant de passer par le plus de branches possibles (conditions, boucles, ...) du programme.

```

20 :      fseek(file,start,SEEK_SET);
20 :      highscoresTag='0';
20 :      highScores=0;
780 :      while(fscanf(file,"%c", &c7)!=EOF){
760 :          fscanf(file,"< Highscores >%d < / Highscores %c", &highScores, &highscoresTag);
760 :          if( (highscoresTag=='>') && (highScores>=4) && ftell(file)<End ) {highscoresflag=1; break;}
20 :      }
20 :      if(width==0 || widthflag==0){
0 :          width=7;
0 :          printf("Incorrect file format, default value of width (7) is loaded\n");
20 :      }
20 :      if(height==0 || heightflag==0){
0 :          height=6;
0 :          printf("Incorrect file format, default value of height (6) is loaded\n");
20 :      }
20 :      if(highScores==0 || highscoresflag==0){
0 :          highScores=5;
0 :          printf("Incorrect file format, default value of highscore list (5) is loaded\n");
20 :      }
20 :      fclose(file);
20 :      (*w) = width;
20 :      (*h) = height;
20 :      (*hs) = highScores;
20 : }

```


Détection de défauts et erreurs

Bien que les tests permettent de trouver les bugs les plus gros dans un programme, on ne peut souvent pas tester tous les états possibles, et il peut rester des bugs arrivant dans certains cas particuliers, ou bien des défauts non directement visibles. On peut alors se tourner vers des outils spécialisés dans la détection d'erreurs. Nous allons ainsi voir deux types d'outils : un débogueur et un outil d'exécution symbolique.

Débogueur

Un débogueur permet de contrôler l'exécution d'un programme et ajoute des informations supplémentaires à ce dernier. On peut ainsi détecter des erreurs comme des accès mémoire non autorisés, ou bien même des fuites mémoire.

Exemple La puissance 4 que nous sommes en train de modifier déclenche des erreurs de segmentation dans certains cas. Nous allons donc utiliser un outil de débogage simple d'utilisation : valgrind. Il suffit d'exécuter notre programme avec valgrind pour que ce dernier nous détaille la provenance de l'erreur :

```
==2528== Invalid read of size 8
==2528==    at 0x408ACF: checkEmpty1 (ia.c:34)
==2528==    by 0x409368: Medium (ia.c:93)
==2528==    by 0x405C66: play_ia (actions.c:20)
==2528==    by 0x402631: main (appli.c:146)
```

On a ainsi accès à la ligne, à la suite d'appel des fonctions et à la taille de l'accès mémoire ayant déclenché l'erreur, il ne reste qu'à la corriger en ajoutant des conditions (voire des assertions) anticipant les accès non autorisés. En vert les corrections ajoutées :

```
int checkEmpty1(Board b, int num, int numOfrow)
{
    assert(num >= 0 && num < b.width);
    assert(numOfrow >= 0 && numOfrow < b.height);
    if (numOfrow == (b.height - 1) && b.tab[numOfrow][num] == VIDE) {
        return 1;
    }
    else if (numOfrow + 1 < b.height && b.tab[numOfrow + 1][num] != VIDE && b.tab[numOfrow][num] == VIDE) {
        return 1;
    }
}
```

```

    }
    return 0;
}

```

A noter que Valgrind nous donne également des informations sur les fuites mémoire en nous informant sur le nombre de blocs alloués/libérés.

```

==2869== HEAP SUMMARY:
==2869==      in use at exit: 42 bytes in 1 blocks
==2869==    total heap usage: 26 allocs, 25 frees , 100,764 bytes allocated
==2869==
==2869== 42 bytes in 1 blocks are definitely lost in loss record 1 of 1
==2869==    at 0x4C2DB8F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==2869==    by 0x406695: board_new (board.c:35)
==2869==    by 0x401609: main (appli.c:21)

```

Exécution symbolique

Cette méthode pour trouver des bugs dans un programme consiste à explorer tous les chemins d'exécution possibles de ce programme. Bien que cette méthode donne de bons résultats, elle a l'inconvénient de ne s'appliquer qu'à de petits codes (ou petites portions de codes) ceci étant dû au nombre de chemins explorés selon la taille du code.

Exemple Nous avons utilisé Klee sur notre programme. Klee nécessite d'instrumenter son code afin de désigner quelles variables peuvent voir leur valeur changer. Dans notre cas, il s'agit des entrées de l'utilisateur.

En lançant Klee et en attendant quelques minutes, on a pu débusquer une erreur d'accès mémoire via le message suivant :

```
KLEE: ERROR: appli.c:641: memory error: out of bound pointer
```

Même une fois l'erreur trouvée, Klee continue à tourner et cherche d'autres problèmes.

Profiling

Dans le but d'optimiser son code, il est possible d'utiliser le profilage de code qui donne des informations sur l'exécution du programme. Un profiling basique consiste à donner le temps passé dans chaque fonction du programme. L'outil gprof permet de faire cela. C'est très intéressant lorsqu'on a un programme qui tourne lentement et qu'on souhaite savoir quelle partie du code optimiser. Si on utilise le profiling sur un programme qui de base s'exécute trop vite, on ne pourra rien en tirer et il faudra donc adapter le programme ou les entrées afin de simuler une exécution longue et ainsi obtenir un profiling exploitable.

Exemple Voici un profiling réalisé avec gprof sur le puissance 4.

Flat profile :

```
Each sample counts as 0.01 seconds.
%   cumulative   self           calls   self   total    name
time  seconds    seconds               us/call  us/call             name
56.46      9.76      9.76             9907    985.34    985.34  board_print
11.97     11.83      2.07            10000    207.04    207.04  diagonal
11.16     13.76      1.93            10000    193.03    193.03  horizontalScore
10.47     15.57      1.81            10000    181.03    181.03  verticalScore
 9.83     17.27      1.70            10000    170.03    170.03  diagonal1
 0.06     17.28      0.01            14906      0.67      0.67  checkfull
 0.06     17.29      0.01              0.00      0.00      0.00  main
 0.00     17.29      0.00            24906      0.00      0.00  board_currentPlayerSymbole
 0.00     17.29      0.00            10000      0.00      0.00  board_put
 0.00     17.29      0.00             9906      0.00      0.00  board_collsFull
 0.00     17.29      0.00             9906      0.00      0.00  checknum
...
```

On trouve tout à droite les noms des fonction du programme étudié. On remarque que les fonction board_print,diagonal,horizontalScore,verticalScore et diagonal1 ont des pourcentages de temps d'exécution plus grand que les autres. On peut alors se demander s'il est possible des les améliorer. Les fonction diagonal,horizontalScore,verticalScore et diagonal1 sont toutes des fonctions fréquemment appelées qui parcourt un tableau de taille NxM afin de calculer un score. Il se trouve qu'on peut facilement améliorer leurs performances en stockant en mémoire le calcul du score précédent. C'est d'ailleurs un règle assez générale: on fait moins de calculs mais on utilise plus de mémoire. En modifiant ainsi les fonctions, on obtient le profiling suivant:

Flat profile :

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self us/call	total us/call	name
99.37	6.10	6.10	9907	615.83	615.83	board_print
0.16	6.11	0.01	40000	0.25	0.25	board_lastPlayRow
0.16	6.12	0.01	14906	0.67	0.67	checkfull
0.16	6.13	0.01	10000	1.00	1.25	diagonal1
0.16	6.14	0.01	10000	1.00	1.25	horizontalScore
0.00	6.14	0.00	80000	0.00	0.00	board_lastPlayCol
0.00	6.14	0.00	24906	0.00	0.00	board_currentPlayerSymbole
0.00	6.14	0.00	10000	0.00	0.00	board_put
0.00	6.14	0.00	10000	0.00	0.25	diagonal
0.00	6.14	0.00	10000	0.00	0.25	verticalScore
0.00	6.14	0.00	9906	0.00	0.00	board_collsFull
0.00	6.14	0.00	9906	0.00	0.00	checknum

En ignorant la fonction `board_print` qui est une fonction d'affichage et qui met donc un certain temps à s'exécuter (on peut le voir dans le `us/call`), on voit qu'on a gagné en pourcentage de temps passé dans les fonctions concernées. Gprof donne aussi d'autre information comme le nombre d'appel des fonction mais aussi le call graph, qui donne le temps passé dans les fonctions et dans ses fils.

Analyse de performance

Au delà de l'aspect programmation, qui se doit d'être affiné le plus possible afin que le programme s'exécute dans un temps minimal, nous avons optimisé le programme afin de minimiser les temps d'accès aux caches du processeur. En effet, une erreur de prédiction de saut ou une donnée non présente dans un cache lors de l'accès fait perdre un temps considérable dans l'exécution du programme, car il faut dans un cas détecter l'erreur de branchement et la corriger, et dans l'autre cas aller chercher la donnée dans la mémoire. Pour optimiser cet aspect, on peut citer deux bonnes pratiques simples :

- Remplacer les séries de "if" par des "switch", car la table de branchement de ce dernier minimise le pourcentage de "branch miss".
- Changer les ordres de parcours des tableaux dans la mesure du possible afin de minimiser cette fois le pourcentage de "cache miss".

Sur un petit programme, ces optimisations peuvent ne pas être visibles et semblées anecdotique mais sur un projet de plus grande envergure, optimiser cet aspect est essentiel.

Exemple Nous avons travaillé avec l'option "Cachegrind" de Valgrind pour détecter et dénombrer le taux de miss. On peut voir sur le tableau suivant un certain nombre d'informations qui représentent le pourcentage de miss des différents caches, ainsi que le nombre concret de miss. Nous pouvons aussi voir le pourcentage de "branch miss" dans les dernières lignes. Ce tableau est extrait d'une exécution de Cachegrind sur la version de base du projet, avant toute modification (hormis celles qui permettent simplement au programme de compiler). Ce tableau se comprend comme ceci :

I fait référence au cache instruction, I1 au cache le plus rapide d'accès et LL (pour "last level") aux caches plus éloignés.

D fait référence au cache de données (Data) et fonctionne sur le même principe.

Les branchements quant à eux sont simplement énoncés à la fin de l'analyse.

```
==4891==
==4891== I   refs :      4,320,762
==4891== I1  misses :      14,181
==4891== LLi misses :       1,990
==4891== I1  miss rate :      0.33%
==4891== LLi miss rate :      0.05%
==4891==
```

```

==4891== D    refs :      1,758,266 (1,140,077 rd + 618,189 wr)
==4891== D1   misses :      3,357 ( 2,709 rd + 648 wr)
==4891== LLd  misses :      2,668 ( 2,086 rd + 582 wr)
==4891== D1   miss rate :      0.2% ( 0.2% + 0.1% )
==4891== LLd  miss rate :      0.2% ( 0.2% + 0.1% )
==4891==
==4891== LL  refs :      17,538 ( 16,890 rd + 648 wr)
==4891== LL  misses :      4,658 ( 4,076 rd + 582 wr)
==4891== LL  miss rate :      0.1% ( 0.1% + 0.1% )
==4891==
==4891== Branches :      804,931 ( 754,035 cond + 50,896 ind)
==4891== Mispredicts :      20,138 ( 19,816 cond + 322 ind)
==4891== Mispred rate :      2.5% ( 2.6% + 0.6% )

```

Figure 1 : Exécution de Cachegrind sur le programme de base.

Après les modifications de notre programme, nous avons obtenu les valeurs suivantes, qui s'interprètent exactement de la même façon.

```

==5578==
==5578== I    refs :      5,222,873
==5578== I1   misses :      2,457
==5578== LLi  misses :      1,762
==5578== I1   miss rate :      0.05%
==5578== LLi  miss rate :      0.03%
==5578==
==5578== D    refs :      2,167,133 (1,304,576 rd + 862,557 wr)
==5578== D1   misses :      6,828 ( 4,225 rd + 2,603 wr)
==5578== LLd  misses :      4,814 ( 2,422 rd + 2,392 wr)
==5578== D1   miss rate :      0.3% ( 0.3% + 0.3% )
==5578== LLd  miss rate :      0.2% ( 0.2% + 0.3% )
==5578==
==5578== LL  refs :      9,285 ( 6,682 rd + 2,603 wr)
==5578== LL  misses :      6,576 ( 4,184 rd + 2,392 wr)
==5578== LL  miss rate :      0.1% ( 0.1% + 0.3% )
==5578==
==5578== Branches :      1,073,634 (1,001,931 cond + 71,703 ind)
==5578== Mispredicts :      24,986 ( 22,529 cond + 2,447 ind)
==5578== Mispred rate :      2.3% ( 2.5% + 3.4% )

```

Figure 2 : Exécution de Cachegrind sur le programme modifié.

Comme on peut le voir, le nombre de "cache miss" à grandement diminué vu que nous sommes passés de 0.33% à 0.05%.

Il serait probablement possible d'améliorer encore ce score mais cela demanderait beaucoup de temps car cela nécessiterait de reprendre un grand pourcentage du code, ce qui ne vaut pas forcément le coup étant donné que 0.05% de miss est un score correct.

Pour ce qui est des prédictions de branchements, elles ont très peu diminuées car les améliorations que nous avons faites sont peu nombreuses, le programme ne se porte pas vraiment à ce genre d'optimisation.

Il est à noter qu'il est normal d'avoir quelques "branch miss" sur un programme de cette taille.

Analyse de vulnérabilité

Outil pour l'analyse de vulnérabilité

Nous avons déjà vu précédemment l'outil KLEE, qui permet de détecter des erreurs automatiquement dans un programme en simulant des traces d'exécution de ce dernier, en essayant d'avoir la plus large couverture de code possible.

Dans ce même but, nous avons utilisé l'outil AFL qui va lui aussi détecter des erreurs via une analyse statique. AFL va, contrairement à KLEE qui modifie directement les variables dans le code, modifier un fichier d'entrées donné par l'utilisateur.

Ce fichier, qui doit couvrir une partie des entrées possible du fichier, va ensuite être transformé par AFL pour trouver des plantages.

La modification est dite "génétique" dans le sens où, même si les modifications sont aléatoires, une modification utile (qui provoque un plantage) sera repérée et alors les transformations suivantes du fichier iront dans le même sens, pour provoquer d'autres potentiels plantages.

Chaque plantage est alors une potentielle faille de sécurité à corriger. Les bugs trouvés sont donc en grande partie des entrées incorrectes.

Les fichiers d'entrées provoquant un plantage sont alors sauvegardés et transmis à l'utilisateur qui peut alors corriger les bugs.

Exemple d'analyse de vulnérabilité

Dans notre puissance 4, l'une des choses lues par le programme est le fichier XML qui donne la taille de la grille. Bien que ce soit potentiellement intéressant de trouver des bugs dans la lecture de ce fichier, nous avons préféré nous focaliser sur le fuzzing des entrées des joueurs du programme. Nous avons donc fait le nécessaire pour que afl prenne ce fichier en paramètre. Nous allons maintenant voir un exemple de l'exécution d'AFL sur le projet de base. Nous pouvons trouver un grand nombre d'informations, et principalement le nombre de crash, crash unique, les chemins empruntés et les timeout.

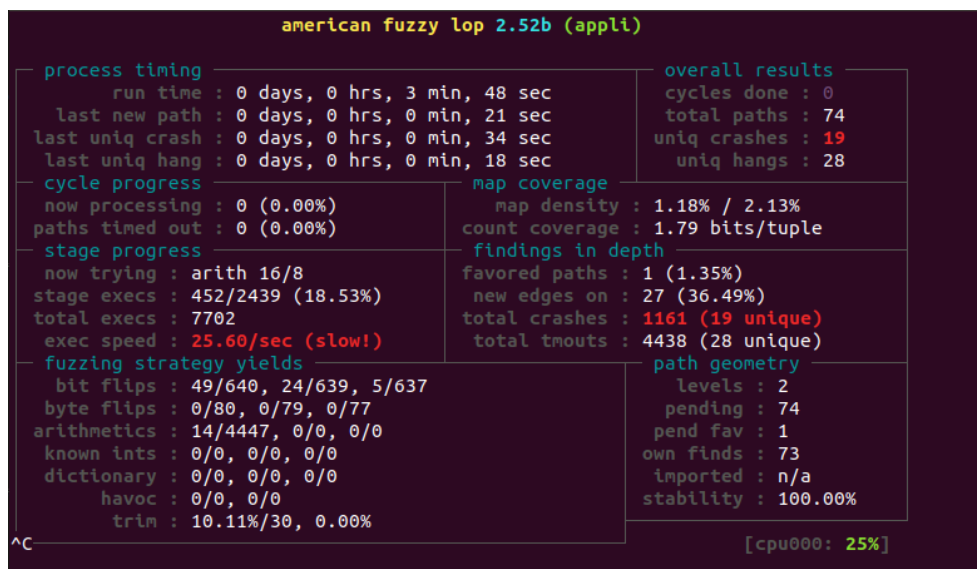


Figure 1 : Exécution d'AFL sur le programme de base.

Nous pouvons voir un très grand nombre de crashes uniques, voici maintenant l'exécution d'AFL sur le projet modifié :

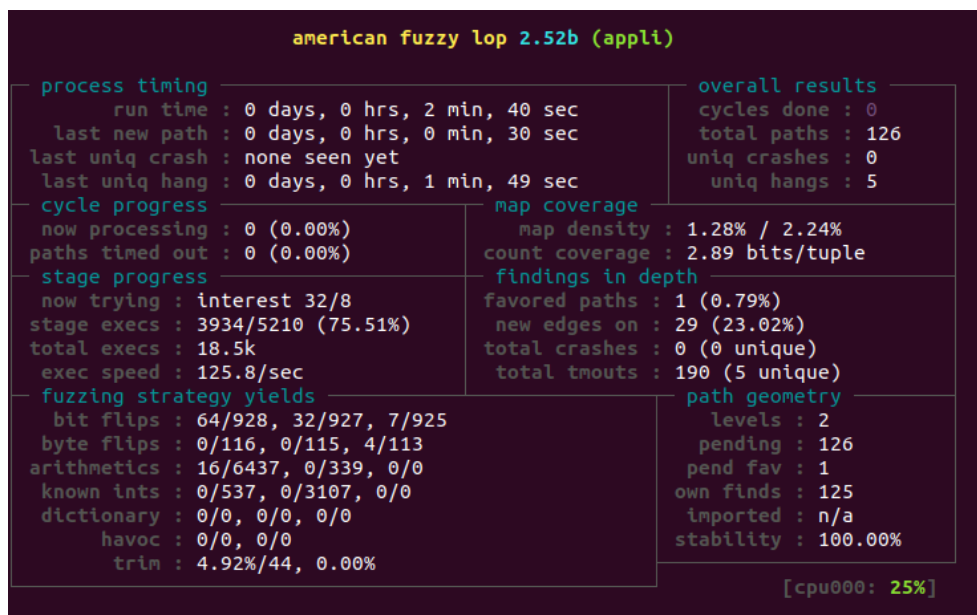


Figure 2 : Exécution d'AFL sur le programme modifié.

Comme nous pouvons le constater, les crashes ont disparus grâce aux mod-

ifications que les différents outils nous ont amené à faire, nous avons résolu certains bug avec AFL mais comme malheureusement il ne peut détecter que les crashes, son utilisation doit se faire en complémentarité avec d'autres outils comme Valgrind.

Conclusion