

# Analyse Statique de Programmes -- TP Frama-C

---

CentraleSupélec

Enseignant: Virgile Prevosto

## Préliminaires

Une image docker est disponible ici: <https://github.com/Frederic-Boulanger-UPS/docker-webtop-3asl> et peut être utilisée soit localement, soit depuis [MyDocker](#). Les fichiers C utilisés pour ce TP sont tous dans le répertoire `config`, qui est monté automatiquement dans le container docker si vous utilisez les scripts associé ([PowerShell](#) pour Windows ou `sh` pour Linux/macOS/BSD). Ces scripts devraient automatiquement ouvrir un onglet de votre navigateur web avec une session IceWM. Si ce n'est pas le cas, vous pouvez le faire manuellement: `http://localhost:3000`

L'image Docker contient Frama-C 29.0, ce que l'on peut vérifier en lançant `frama-c -version` dans un shell.

Dans ce TP, nous allons essentiellement utiliser le greffon Eva. Lancer l'analyse se fait en passant l'option `-eva` à Frama-C. L'ensemble des options du greffon peut se retrouver avec `frama-c -eva-help`. Le manuel du greffon est également disponible sur [le site de Frama-C](#). Notez toutefois que les options nécessaires pour réaliser le TP seront mentionnées dans l'énoncé.

## CERT C EXP33-C. Do not read uninitialized memory

On s'intéresse dans cet exercice à un exemple tiré des [règles de codage C du CERT](#) américain. En l'occurrence, il s'agit d'illustrer la règle [EXP33-C](#), qui rappelle qu'il est important de s'assurer que toute location mémoire dans laquelle on lit a été préalablement initialisée.

L'exemple est reproduit ci-dessous, et se trouve également dans le fichier `cert_exp_33.c`.

```
#include <stddef.h>

void set_flag(int number, int *sign_flag) {
    if (NULL == sign_flag) {
        return;
    }

    if (number > 0) {
        *sign_flag = 1;
    } else if (number < 0) {
        *sign_flag = -1;
    }
}

int is_negative(int number) {
    int sign;
    set_flag(number, &sign);
}
```

```

    return sign < 0;
}

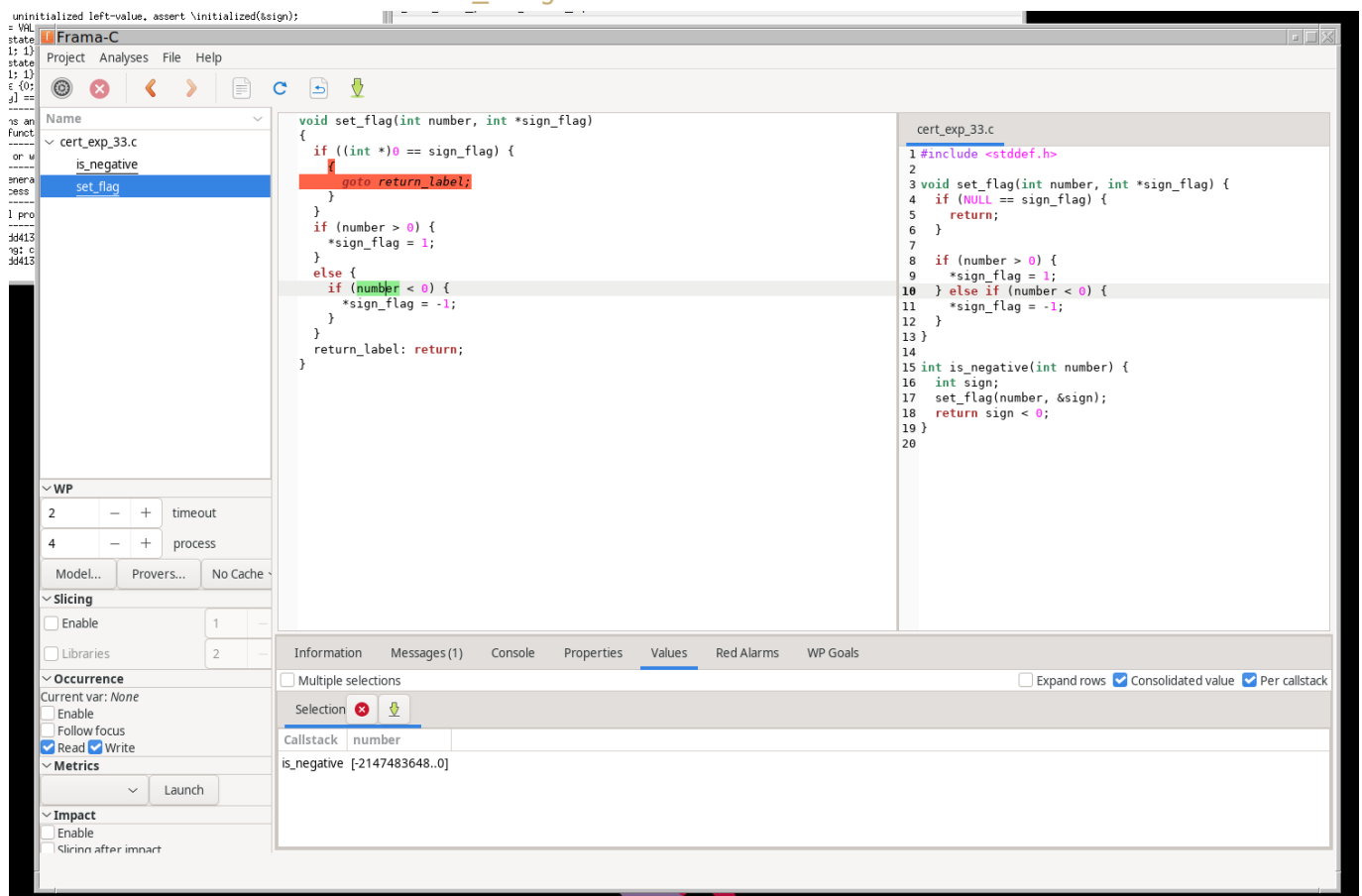
```

## Première exécution de Frama-C

Le fragment de code fourni n'est pas une application complète. Il va donc falloir indiquer à Frama-C quel est le point d'entrée de notre analyse. Lancer `frama-c -eva -main is_negative cert_exp_33.c`. Quels sont les messages émis? Lequel correspond au problème évoqué sur le site du CERT?

Lancer ensuite l'interface graphique: `frama-c-gui -eva -main is_negative cert_exp_33.c`, et remarquer l'assertion ACSL générée par Eva au dessus de l'instruction contenant potentiellement un comportement indéfini. Regarder également l'onglet **Red Alarms**, qui indique s'il existe des alarmes se produisant à coup sûr pour au moins un état abstrait (mais pas nécessairement pour tous les états atteignant le même point de programme). Vérifier qu'il n'y a pas de telle alarme.

Utiliser l'onglet **Values** de l'interface graphique pour déterminer la valeur d'expressions en différents points du programme en cliquant sur les expressions en question comme sur l'exemple ci-dessous pour `number` dans le deuxième `if` de `set_flag`:



Dans la suite du TP, on proposera préférentiellement d'utiliser l'interface textuelle, mais il est toujours possible d'utiliser les mêmes options avec l'interface graphique. Le contenu affiché sur le terminal dans le premier cas se trouve alors dans l'onglet **Console** de l'interface graphique.

## Afficher des valeurs en cours d'analyse

L'alarme levée par Eva n'est que potentielle: on ne sait pas quelles valeurs de `number` pourrait y conduire, ni même s'il ne s'agit pas en fait d'une *fausse alarme*. Pour aller un peu plus loin, nous allons demander à

Eva d'effectuer une analyse plus précise. Tout d'abord, afin de mieux voir comment s'effectue la propagation des valeurs abstraites, nous allons ajouter des appels à des fonctions

`Frama_C_show_each_*`: chaque fois qu'Eva atteint un de ces appels, l'analyseur affichera la valeur abstraite calculée pour chacun des arguments fournis.

Ajouter l'instruction `Frama_C_show_each_set_flag(number, *sign_flag);` à la fin de la fonction `set_flag`, et `Frama_C_show_each_is_negative(number, sign);` après l'appel à `set_flag` dans `is_negative`, puis lancer `frama-c` avec les mêmes arguments que précédemment et observer l'affichage obtenu.

## Partitionner les traces

Un des moyens d'améliorer la précision de l'analyse consiste à utiliser le *partitionnement de traces*, c'est à dire à ne pas systématiquement fusionner les états abstraits provenant de différentes branches, mais à propager simultanément plusieurs états abstraits, suivant certains critères. Eva propose notamment l'option `-eva-partition-history`, qui prend un entier `n` en argument. Lorsque cette option est activée, les traces qui diffèrent sur au moins 1 des `n` dernières conditions rencontrées seront gardées séparées.

Lancer Eva en ajoutant cette option, pour différentes valeurs de `n`. Que constatez-vous sur les affichages provoqués par les fonctions `Frama_C_show_each*`?

## Partitionnement inter-procédural

Par défaut, le partitionnement est *intra*-procédural, c'est à dire qu'au moment où une fonction où du partitionnement s'est produit retourne, on fusionne les états partitionnés en un seul état abstrait qui est remonté à l'appelant. Si on veut conserver le partitionnement au niveau de l'appelant, il faut utiliser `-eva-interprocedural-history`.

Utiliser cette option. Que constatez-vous sur les messages d'Eva? Lancer l'interface graphique avec ces options et vérifier qu'il y a bien une alarme rouge

## Analyse du code corrigé

Utiliser la version de `set_flag` corrigée telle que proposée sur la page, et vérifier qu'aucune alarme n'est rapportée par Eva.

## CERT C EXP33-C bis

On s'intéresse maintenant à un autre exemple d'utilisation de mémoire non initialisée, cette fois-ci par le biais de la fonction `realloc`. Le code sur lequel on va travailler se trouve dans le fichier `cert_exp_33_realloc.c`, et est reproduit ci-dessous:

```
#include <stdlib.h>
#include <stdio.h>
enum { OLD_SIZE = 10, NEW_SIZE = 20 };

int *resize_array(int *array, size_t count) {
    if (0 == count) {
        return 0;
    }
}
```

```

int *ret = (int *)realloc(array, count * sizeof(int));
if (!ret) {
    free(array);
    return 0;
}

return ret;
}

void func(void) {

    int *array = (int *)malloc(OLD_SIZE * sizeof(int));
    if (0 == array) {
        /* Handle error */
        return;
    }

    for (size_t i = 0; i < OLD_SIZE; ++i) {
        array[i] = i;
    }

    array = resize_array(array, NEW_SIZE);
    if (0 == array) {
        /* Handle error */
        return;
    }

    for (size_t i = 0; i < NEW_SIZE; ++i) {
        printf("%d ", array[i]);
    }
}

```

## Première analyse

Lancer Eva en utilisant le point d'entrée le plus approprié. Quelles sont les alarmes émises? Quelle est celle qui correspond à la lecture d'une location mémoire non-initialisée?

## Augmenter la précision des boucles

Comme dans l'exercice précédent, Eva signale une alarme potentielle, mais on ne sait pas s'il s'agit d'une vraie alarme ou d'une fausse. Pour faire apparaître une alarme rouge, il faut améliorer la précision de l'analyse. Ici, ce sont les deux boucles de `func` qui posent problème. Plusieurs possibilités existent pour demander à Eva de garder séparés les états correspondant aux différents tours de boucle. On peut notamment:

- utiliser l'option `-eva-min-loop-unroll <n>` pour demander à Eva de garder séparés les `n` premiers tours avant de commencer à joindre (et éventuellement élargir) les états;
- ajouter des annotations `/*@ loop unroll <n>; */` avant les boucles, pour obtenir le même effet, mais pour des boucles particulières;

- utiliser l'option `-eva-slevel <n>` pour demander à Eva de propager jusqu'à `n` états distincts pour chaque point de programme. Contrairement aux options précédentes, cela s'applique à tous les points de programme, donc les `if` vont aussi provoquer des séparations d'état;
- utiliser l'option `-eva-precision <n>`, qui positionne un certain nombre d'options améliorant la précision, dont `min-loop-unroll` et `slevel`. `n` peut aller de `0` (le moins précis) à `11` (le plus précis).

Expérimenter au moins une de ces options pour faire apparaître une alarme rouge pour l'initialisation. Vous pouvez utiliser des appels à `Frama_C_show_each_*` dans les boucles pour voir comment la propagation des états est affectée par ces options.

## Analyse précise de `realloc`

L'autre alarme générée par Eva provient du fait qu'avec les options par défaut, Eva ne conserve pas la relation entre la valeur de retour de `realloc` et le statut de son premier argument (plus précisément, `realloc` libère la mémoire du pointeur passé en argument si et seulement si l'allocation réussit, c'est à dire qu'elle retourne un pointeur non-NULL). De ce fait, l'analyseur ne peut garantir que le `free(array)` est correct. Si on a utilisé l'option `-eva-slevel` (ou `eva-precision`) à la question précédente, le problème disparaît, car les deux états possibles en retour de `realloc` sont gardés séparés. Sinon, il faut forcer Eva à faire cette séparation. Pour cela, on peut utiliser l'annotation `//@ split ret==0;`, juste après l'appel à `realloc`.

Après avoir ajouté cette annotation, vérifier que l'alarme sur `free(array)` disparaît bien, même en utilisant `-eva-min-loop-unroll` ou `/*@ loop unroll <n>; */`

## Analyse de la version corrigée

Ajouter la [correction proposée](#) par le CERT. Pour cela, on ajoute un argument `old_count` à `resize_array`, qui contient la taille initiale de `array`, et si `old_count` est plus petit que `count`, on utilise `memset` (déclarée dans l'en-tête `string.h`) pour initialiser les nouvelles cases à `0`.

Utiliser Eva pour vérifier qu'il n'y a pas d'erreur possible à l'exécution.

## CWE 20

Cet exercice reprend un exemple de vulnérabilité liée à la mauvaise validation des entrées utilisateurs, plus spécifiquement la classe [CWE20](#) de la *Common Weakness Enumeration*. Le code est fourni dans le fichier `cwe20.c` et ci-dessous.

```
#include <stdio.h>
#include <stdlib.h>
void die(const char* s) {
    printf("%s", s);
    exit(2);
}

#define MAX_DIM 15

typedef int board_square_t;
```

```

/* board dimensions */

int main () {
    int m,n, error;
    board_square_t *board;
    printf("Please specify the board height: \n");
    error = scanf("%d", &m);
    if ( EOF == error ){
        die("No integer passed: Die evil hacker!\n");
    }
    printf("Please specify the board width: \n");
    error = scanf("%d", &n);
    if ( EOF == error ){
        die("No integer passed: Die evil hacker!\n");
    }
    if ( m > MAX_DIM || n > MAX_DIM ) {
        die("Value too large: Die evil hacker!\n");
    }
    board = (board_square_t*) malloc( m * n * sizeof(board_square_t));
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            board[n*i+j] = 0;
        }
    }
    if (board[(m-1)*n] == 0) return 0; else return 1;
}

```

## Première analyse

Lancer Frama-C/Eva sur le code. Que peut-on dire des alarmes émises?

### Patch 1

Modifier le code pour éviter le problème décrit dans la page de la CWE20, et relancer l'analyse. Modifier éventuellement les options de précision. Peut-on supprimer toutes les alarmes restantes?

### Patch 2

Modifier le programme pour qu'on teste si l'allocation a réussi avant d'utiliser le bloc mémoire correspondant (si l'allocation échoue, on retourne directement avec la valeur 2), puis relancer l'analyse et regarder si on peut effectivement éviter les alarmes.

## Améliorer la précision

Le problème vient du fait que Eva ne garde pas de lien entre la taille du bloc alloué par `malloc` et les valeurs de `m` et `n`. Pour pallier à ce problème, on peut partitionner l'analyse, via deux directives `//@ split`, de manière à ce que pour chaque élément de la partition considère une valeur précise de `m` et de `n`. Il conviendra aussi d'utiliser l'option `-eva-alloc-builtin fresh` pour que Eva utilise des bases différentes pour chaque élément de la partition.

## libstring

On s'intéresse à la bibliothèque de manipulation de chaînes [libstring](#), dont les deux principaux fichiers sont [libstring.c](#) et [libstring.h](#).

## concat

À l'aide de la fonction `Frama_C_interval` de `__fc_builtin.h` qui prend en entrée deux entiers et renvoie un entier au hasard compris entre ces bornes (ce qui sera donc interprété par Eva comme un intervalle), écrire une fonction `main` qui initialise deux `string_t` à partir de deux chaînes C de 255 caractères aléatoires (entre 1 et 127) chacune (on n'oubliera pas le 0 terminal de chaque chaîne C), puis en réalise la concaténation, et imprime le résultat.

Analyser avec Frama-C/Eva ce programme et vérifier l'absence d'erreur à l'exécution.

## replace

Définir de même un contexte aussi générique que possible (mais avec des chaînes de taille fixe et relativement réduite) pour étudier avec Eva le comportement de la fonction `string_replace`. Corriger le code si besoin.

## Miniz

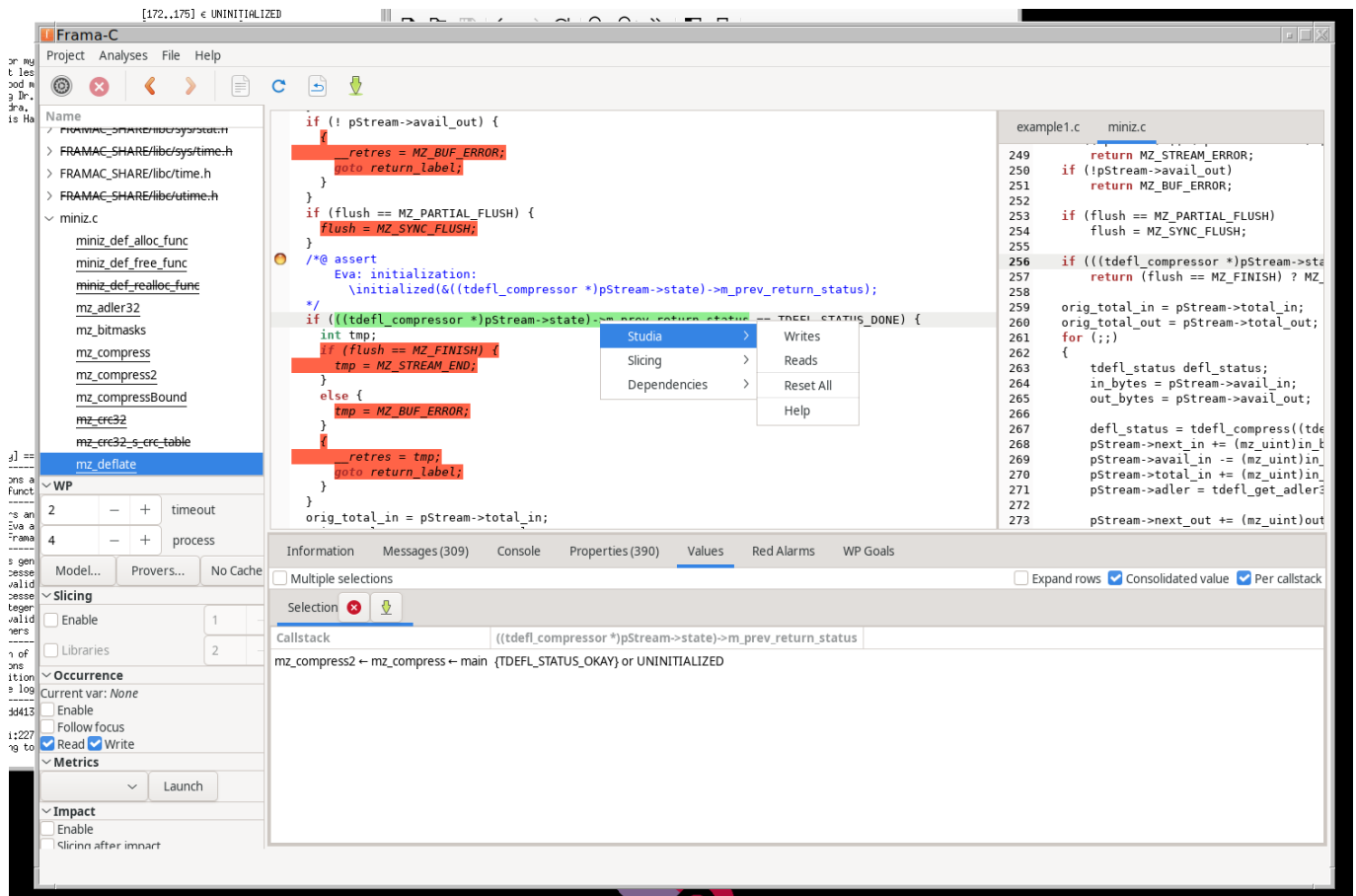
On s'intéresse dans cet exercice à une bibliothèque de compression, [miniz](#), et plus précisément à sa dernière release stable, [3.0.2](#). Les fichiers qui vont nous intéresser sont [miniz.h](#), [miniz.c](#) et [example1.c](#).

## Compression

### Première analyse

lancer Eva sur l'exemple: `frama-c -eva miniz.c example1.c`. Que peut-on dire du résultat?

Pour des programmes de taille raisonnable, il est généralement nécessaire d'utiliser l'interface graphique pour pouvoir naviguer dans le code. En particulier, le greffon Studia permet d'identifier les endroits où on a écrit pour la dernière fois dans une location mémoire avant le point de programme courant. On l'active par un clic droit sur la location qui nous intéresse, ainsi que montré sur l'image ci-dessous:



En partant de la première alarme émise par Eva (qui sont listées dans l'onglet Message de la GUI), identifier l'origine de la perte de précision

## Améliorer la précision

En utilisant judicieusement les options `-eva-slevel <n>` qui permet de garder `n` états séparés, `-eva-partition-history <m>`, qui permet de garder séparées les `m` dernières branches de l'analyse, et `-eva-split-return-function f:m` qui permet de séparer les états de retour d'une fonction `f` en fonction de sa valeur de retour (plus précisément si elle retourne `m` ou autre chose), Réduisez le nombre d'alarmes émises par Eva. On pourra bien entendu également utiliser les options vues dans les exercices précédents. Arrive-t-on à supprimer toutes les alarmes?

## Patch

Les dernières alarmes notent un vrai problème dans le code. Toujours à l'aide de Studia, identifiez le(s) point(s) où on devrait initialiser les locations en cause, et vérifier qu'on n'a plus d'alarme dans le code ainsi modifié.