L3-MI / PROG
TP#5 : Valgrind & ASan

2022-W41

Présentation

Le but de ce TP est de vous familiariser avec deux outils d'analyse dynamique, Valgrind et ASan, permettant de détecter un certain nombre de bugs (notamment liés à la gestion et aux accès mémoire). Ces deux outils sont quelque peu complémentaires, car ils peuvent chacun détecter des bugs difficilement (ou non-) détectables par l'autre. Nous les présentons ici très brièvement.

Valgrind

Valgrind [1] est un analyseur dynamique capable de détecter des bugs d'accès mémoire illégaux, de lecture de mémoire non initialisée, de désallocation multiple, etc.. Le principe de base de Valgrind consiste à exécuter le programme que l'on souhaite analyser dans un environnement virtualisé qui maintient ses propres informations sur l'espace mémoire et son contenu.

Valgrind s'utilise typiquement en :

1. compilant son programme avec un niveau d'optimisation « raisonnable » (par ex. `-O1`) et les options facilitant le débuggage (typiquement `-g -fno-omit-frame-pointer`) ;
2. remplaçant l'invocation du programme `$ prog <options>` (ici depuis un shell) par `$ valgrind prog <options>`.

ASan

L'*AdressSanitizer* "ASan" [2] est un analyseur dynamique capable de détecter des bugs d'accès mémoire illégaux, d'accès à de la mémoire déjà désallouée, etc.. Le principe de base d'ASan consiste à modifier (à « instrumenter ») le programme lors de la compilation afin d'ajouter un environnement de détection et des tests associés.

ASan s'utilise typiquement en :

1. compilant son programme avec un niveau d'optimisation « raisonnable » (par ex. `-O1`) et les options facilitant le débuggage (typiquement `-g -fno-omit-frame-pointer`) ;
2. ajoutant `-fsanitize=address` aux options de compilation (pour un compilateur compatible, par ex. les versions suffisamment récentes de clang ou gcc), à la fois pour la production de fichiers objets `.o` et pour la phase de *link*.

Par ailleurs, il existe également un *UndefinedBehaviorSanitizer* "UBSan" de fonctionnement similaire, qui s'utilise en ajoutant `-fsanitize=undefined` aux options de compilation.

1. https://valgrind.org/
2. https://github.com/google/sanitizers/wiki/AddressSanitizer

Comparaison

On peut comparer (très brièvement) Valgrind et ASan en soulignant les aspects suivants :
— Certains bugs sont mieux détectés avec un outil en particulier. Par exemple ASan ne détecte pas la lecture de mémoire non initialisée, mais il détecte certains accès mémoire illégaux mieux que Valgrind (notamment les « petits » dépassements de tableau).
— ASan est assez rapide (il dégrade peu les performances du programme instrumenté) et s'intègre facilement au processus de compilation. Il nécessite cependant un accès aux sources du programme.
— Le test d'un programme sous Valgrind peut être assez lent, mais est possible pour tout exécutable sans nécessiter un accès aux sources ou une recompilation ; il n'est pas non plus restreint aux programmes écrits en C ou C++.

Conseils

De façon générale (hors du cadre spécifique de ce TP), on peut formuler les conseils suivants : 1) systématiquement utiliser ASan et UBSan lors du processus de développement (sans attendre la présence visible d'un bug) ; 2) en cas de bug observé non détecté par ASan/UBSan, utiliser Valgrind (mieux vaut dans ce cas désactiver ASan et UBSan).

Exercice 1

Commencez par extraire le programme `prpnbs.c` de l'archive https://membres-ljk.imag.fr/Pierre.Karpman/pc2022_tp5.tar.bz2, et compilez-le tout d'abord sans options particulières.

Q.1 :

1. Lancez le programme successivement avec les arguments suivants : 100, 1 000, 10 000, 100 000, 1 000 000, pour la fonction `prpnbs`.
2. Faites de même (avec argument 100) en utilisant une fois Valgrind et une fois ASan. Expliquez les messages d'erreur, et corrigez le bug.

Q.2 :

1. Lancez le programme avec argument 3 000 000 000 (si vous avez suffisamment de mémoire). Que se passe-t il ?
2. Faites de même avec UBSan, et corrigez le bug (sans changer la signature de la fonction).

Q.3⋆ :

1. Quel est l'algorithme antique implémenté par ce programme, et que fait il ?
2. Expliquez l'intérêt des instructions `nums -= 2` et `nums += 2` aux lignes 15 et 40.
3. Faites une analyse de coût (on pourra utiliser $H_n := \sum_{k=1}^{n} 1/k \in \Theta(\log(n))$).

Exercice 2

Commencez par extraire le programme `minesweeper.c` de l'archive https://membres-ljk.imag.fr/Pierre.Karpman/pc2022_tp5.tar.bz2.

Q.1 : En utilisant Valgrind, ASan et UBSan, trouvez et corrigez les (au moins) 6 bugs présents dans ce programme. (On conseille de tester le programme avec (entre autres) des petites tailles de grille "*mapsize*", par exemple 3.)

Exercice 3

Commencez par extraire le programme `rf.c` de l'archive https://membres-ljk.imag.fr/Pierre.Karpman/pc2022_tp5.tar.bz2.

Q.1 :

1. *Sans exécuter le programme*, pouvez-vous prévoir quel problème risque de survenir à l'exécution ?
2. Vérifiez votre hypothèse en exécutant le programme et en utilisant Valgrind et/ou ASan.