INF889A

Analyse de programmes pour la sécurité logicielle

Chapitre 0 - Introduction à l'analyse de programme

Quentin Stiévenart

Hiver 2024

Présentation du cours

Site du cours

Tout le contenu est disponible sur https://inf889a.uqam.ca.

Prérequis

- Savoir programmer
  - Connaissance du C++ recommandée
- Fondements de sécurité
  - INF4471 Introduction à la sécurité informatique
  - INF600C Sécurité des logiciels et exploitation de vulnérabilités

Structure du cours

- Théorie et mise en pratique
- Intra
- Présentations et projets

Calendrier

Voir le site du cours

Structure du cours

Théorie

- Chapitre 0 : Introduction à l'analyse de programmes
- Chapitre 1 : Fuzzing
- Chapitre 2 : Représentation intermédiaire et instrumentation
- Chapitre 3 : Exécution symbolique
- Chapitre 4 : Analyse de flot de données
- Chapitre 5 : Contrôle de flot d'information
- Chapitre 6 : Interprétation abstraite

Structure du cours

Séminaire

- Présentation d'un article et d'un outil
- Présentation d'un projet de session

Évaluation

Évaluation	Pondération	Échéance
Examen intra	30%	Semaine 10
Présentation d'article et d'outil	30%	Semaine 12-13
Projet de session	30%	Semaine 14-15
Participation au cours	10%	–

Matière de l'intra : les 6 chapitres de théorie

Évaluation

Présentation d'article et d'outil

- Choix d'article et d'outil : semaine 10 au plus tard
- Présentation (2×10%) : semaines 12 et 13
- Mini-rapport (2×5%) : semaine 15

Projet de session

- Choix de sujet (5%) : avant la semaine 6
- Rencontre d'étape (5%) : semaine 9
- Présentation (10%) : semaine 14-15
- Rapport (10%) : semaine 15

Références

- The Fuzzing Book: Tools and Techniques for Generating Software Tests, Zeller, A., Gopinath, R., Böhme, M., Fraser, G., & Holler, C.
  - Pour le chapitre 1
- Program Analysis, cours de Michael Pradel de 2020-2021.
  - Pour les chapitre 3 et 5
- Static program analysis, Møller, A., & Schwartzbach, M. I.
  - Pour les chapitres 4 et 6

Autres références utilisées pour la création du contenu :

- Cours INF889A (Hiver 2020) par Jean Privat
- Cours *Software Quality Analysis* par Coen De Roover (VUB, Belgique)

Introduction à l'analyse de programmes

Analyse de programmes : idée générale

On veut :

- déterminer de façon **automatique** (avec des outils)
- des **propriétés** de programmes
- à partir d'**artefacts** (code source, trace d'exécutions, etc.)

Domaines connexes

Informatique théorique

- Théorie des langages, calculabilité
- Théorie des treillis (*lattice*)
- Logique formelle (solveurs, contraintes, preuves)

Compilation

- Transformation de code
- Sémantique des langages de programmation
- Analyses, optimisations

Domaines connexes

Génie logiciel

- Compréhension de programmes, rétro-ingénérie
- Détection et correction de défauts, qualité logicielle
- Refactorisation et transformation

Cybersécurité logicielle

- Détection de bogues de sécurité
- Détection de logiciels malveillants
- Protection d'applications

Qualité d'une analyse

La mesure de la qualité d'une analyse dépend de la propriété analysée.

Ici, par exemple pour un outil de détection de bogues.

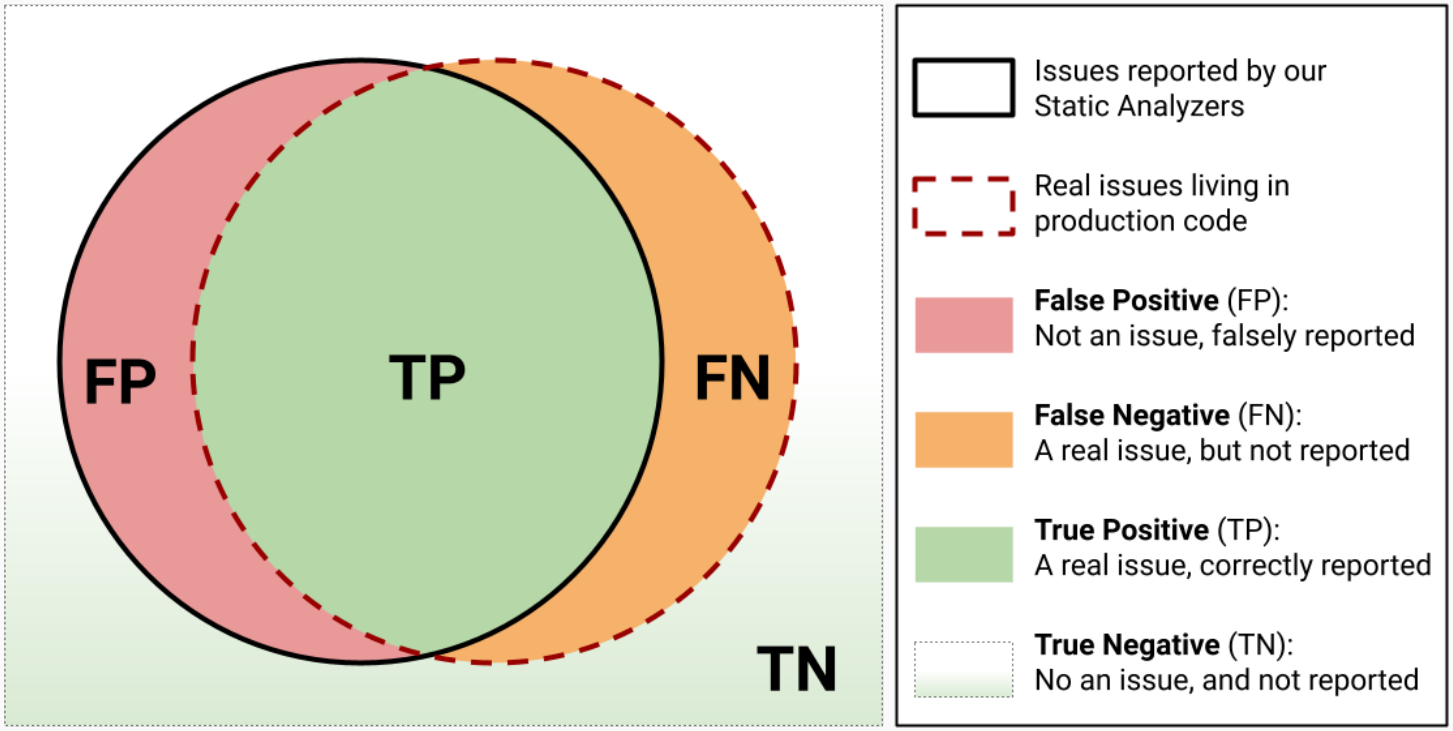
Garanties de l'analyse

- sûreté (*soundness*) : tous les bugs sont trouvés
  - on peut faire "confiance" aux résultats de l'analyse
  - pas de bug trouvé : absence de bugs dans le programme
  - mais peut-être qu'on trouve des "faux" bugs
- complétude (*completeness*) : tous les bugs trouvés sont valides
  - pas de bug trouvé : peut-être qu'il y en a quand même

Ces termes sont utilisés différement selon les domaines et personnes : on préférera la notion de vrais/faux positifs/négatifs.

Classification des résultats : détection de bugs

- Vrais positifs : bugs trouvés
- Faux positifs : bugs détectés mais inexistants
- Faux négatifs : bugs existants mais pas trouvés
- Vrais négatifs : tout le reste



Source

Limites d'une analyse : théorème de Rice

Toute propriété sémantique non triviale d'un programme est indécidable. – Rice (1953)

- Il est fondamentalement impossible d'avoir une analyse à la fois *sound* et *complete*
  - il faudra accepter les faux positifs et/ou faux négatifs

Approximations

On parlera plutôt d'approximation.

- Sur-approximation : on détecte *au moins* tous les bugs
  - mais on en détecte peut-être trop
  - inclus des faux positifs mais pas de faux négatifs
  - c'est une analyse *sound*
- Sous-approximation : les bugs détectés sont tous réels
  - mais on en rate peut-être
  - inclus des faux négatifs mais pas de faux positifs
  - c'est une analyse *complete*

Soundiness

En pratique : un mélange des deux, à cause de :

- fonctionnalités avancées des langages
- taille des programmes

Livshits, et al. (2015). In Defense of Soundiness: A Manifesto. CACM.

Language	Examples of commonly ignored features	Consequences of not modeling these features
C/C++	`setjmp/longjmp` ignored	ignores arbitrary side-effects to the program heap
	effects of pointer arithmetic	
	"manufactured" pointers	
Java/C#	Reflection	can render much of the codebase invisible for analysis
	JNI	"invisible" code may create invisible side-effects in programs
JavaScript	eval, dynamic code loading	missing execution
	data flow through the DOM	missing data flow in program

Astrée : une analyse *sound*

> *Astrée analyzes structured C programs, with complex memory usages, but* ***without dynamic memory allocation and recursion***

Pour qu'Astrée soit *sound* (pas de faux positifs), le choix est fait de ne pas supporter certaines fonctionnalités.

- utilisé pour des systèmes critiques (avions, véhicules spatiaux, réacteurs nucléaires)

Type d'analyse

- Analyse statique : analyse le code sans l'exécuter
- Analyse dynamique : analyse le comportement du code à travers son exécution

Chaque analyse a des garanties et propriétés différentes

Exemple : Analyse statique de Clang

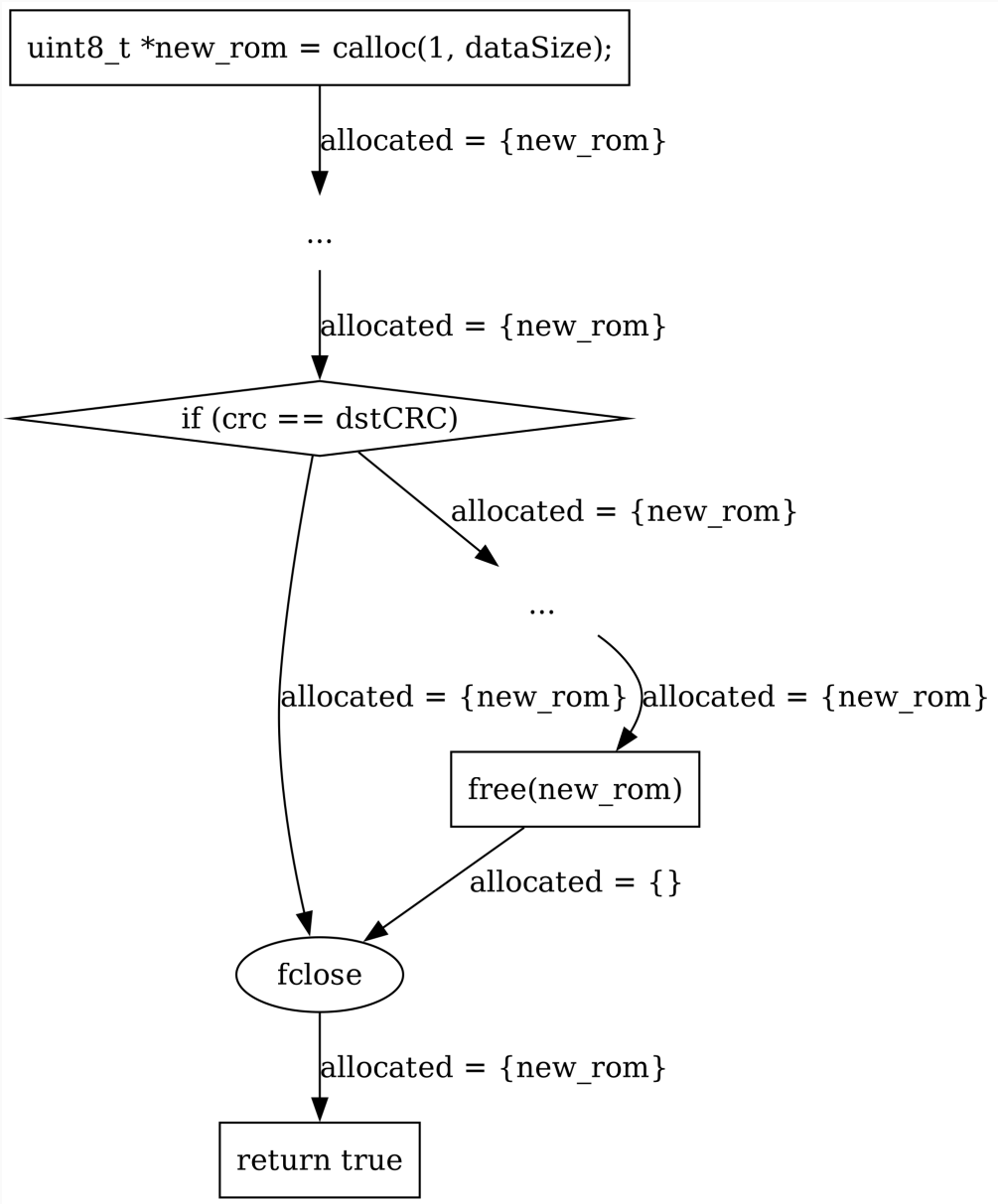
```
$ git clone https://github.com/visualboyadvance-m/visualboyadvance-m
$ cd visualboyadvance-m
$ clang++ --analyze src/**/*

...
src/common/Patch.cpp:412:5: warning: Potential leak
  of memory pointed to by 'new_rom' [unix.Malloc]
    fclose(f);
...
```

Exemple : Analyse statique de Clang

Le code en question de `Patch.cpp` :

```
uint8_t* new_rom = (uint8_t*)calloc(1, dataSize);
...
if(crc == dstCRC) {
    if (dataSize > *size) {
        *rom = (uint8_t*)realloc(*rom, dataSize);
    }
    memcpy(*rom, new_rom, dataSize);
    *size = dataSize;
    free(new_rom);
}
fclose(f);
return true;
```

Analyse statique : comment ça fonctionne ?



Analyse statique : comment ça fonctionne ?

On a modélisé l'exécution du programme :

- `malloc` alloue une région, qu'on retient
- `free` libère une région
- si on a un désaccord à un *merge point*, on a une fuite potentielle

CVE-2013-4559

Trouvée par Clang Static Analyzer, dans `lighttpd1.4/src/server.c`.

Le code suivant est exécuté par `root` :

```
int main (int argc, char **argv) {
    ...
    /* drop root privs */
    if (NULL != pwd) {
        setuid(pwd->pw_uid);
    }
    ... /* setup workers */
}
```

Clang Static Analyzer : CVE-2013-4559

Comment le détecter ?

- on cherche un appel à `setuid`
- on “suit” la valeur de retour
- s’il n’y a pas de check, on émet un warning

Analyse statique : techniques

- Linters, `grep`
  - Basique et limité, mais facile à mettre en place
- Exécution symbolique statique (Chapitre 3)
- Analyse dataflow (Chapitres 4, 5)
- Interprétation abstraite (Chapitre 6)
- Model checking (pas dans ce cours)
- Vérification formelle (pas dans ce cours)

Analyse statique : avantages et limites

- Possibilité de ne pas avoir de faux négatifs (être *sound*)
  - Mais difficile en pratique
- Pas nécessaire d'exécuter le programme
  - Facile à mettre en place
- Passage à l'échelle
  - Facile pour des analyses simples
  - Problème conceptuel pour des analyses plus complexes

Exemple : Analyse dynamique par fuzzing avec AFL++

```
git clone https://github.com/deltabeard/Peanut-GB/
```

Il faut exécuter le programme, idéalement avec un test qui ne prends pas trop de temps : `test/test_external_rom.c`

- On utilise une ROM minimale
- On limite le nombre de frames calculées (par exemple, 100)

```
$ cd Peanut-GB/test && make test_external_rom
$ mkdir -p input/ && cp /path/to/rom.gb input/
$ ./test_external_rom input/rom.gb 100
```

Exemple : Analyse dynamique par fuzzing avec AFL++

Installation

```
docker pull aflplusplus/aflplusplus
cd Peanut-GB/
docker run -ti -v .:/src aflplusplus/aflplusplus
cd /src/test
```

Exemple : Analyse dynamique par fuzzing avec AFL++

Compilation

```
AFL_USE_ASAN=1 CC=afl-clang-fast make test_external_rom
```

- Utilise *Adress Sanitizer* pour détecter les erreurs mémoires
- Utilise une variante de `clang` qui instrumente le code

Fuzzing

```
afl-fuzz -i input/ -o results -- ./test_external_rom @@ 10
```

- `input/` contient nos tests initiaux
- `@@` sera remplacé par le chemin vers notre test
- inspecter `results/` pour voir les résultats

Exemple : CVE trouvées par AFL

> *AFL has discovered a huge number of bugs in all sorts of projects from compilers to image processing libraries.*

331 CVEs listées sur https://github.com/mrash/afl-cve

Fuzzing : comment ça marche ?

- On *instrumente* le programme
  - pour détecter des erreurs mémoires
  - pour mesurer quelles branches sont explorées
- On a un corpus initial d'entrées
- Le fuzzer exécute le programme sur notre corpus, en effectuant des modifications mineures à l'entrée
- Si le programme plante, on a trouvé un bug potentiel

Analyse dynamique : techniques

- Tests
- Fuzzing (Chapitre 1)
- Instrumentation (Chapitre 2)
- Exécution symbolique dynamique (Chapitre 3)

Analyse dynamique : avantages et limites

- Pas de faux positifs
  - On *observe* les bugs concrets
- Pas d'information sur ce qui n'est pas exécuté
- Il faut exécuter le programme
  - Nécéssite un point d'entrée
  - Dépendance à l'environnement d'exécution

Concepts clés

- But : analyser automatiquement des propriétés de programmes
- Qualité d'une analyse
  - Quelles garanties ? *soundness*, *completeness*
  - Théorème de Rice : impossible de combiner *soundness* et *completeness*
  - Vrais positifs, faux positifs, vrais négatifs, faux négatifs
- Utilisation d'approximations
- Analyse statique vs. dynamique