

Introduction à l'Analyse de Code pour la Sûreté et la Sécurité (ACSS)

Sylvain Boulmé et Marie-Laure Potet

<https://boulmes.pages.ensimag.fr/acss>

Fil conducteur de ACSS

1. Débordements arithmétiques ou de tableaux (*buffer overflows*)
↪ **nombreux** bogues critiques pour sûreté ou sécurité.
2. Preuve formelle, vérifiée par ordinateur, d'absence de tels bogues : puissant, mais exige beaucoup d'annotations complexes des programmeur-es (cf. TP).
3. Analyse de code par interprétation abstraite : tout automatique, par "*approximation correcte*" des preuves formelles (mais perte de précision).
4. Applications en compilation : optimisations du code généré.
5. Applications en sécurité : recherches de vulnérabilités en combinaison avec d'autres techniques (i.e. exécution symbolique, vérifications dynamiques) quitte à relâcher la contrainte de correction. (cf. TPs)
6. Les langages de programmation peuvent-ils aider ?

Plan du Cours 1

Les bogues logiciels : conséquences, causes et contre-mesures

Introduction aux méthodes formelles du génie logiciel

Quelques exemples de bogues logiciels tristement célèbres

- ▶ Explosion d'ARIANE 5 en 1996 suite à un débordement arithmétique (370M\$).
- ▶ SQL Slammer - débordements en écriture incontrôlés dans Microsoft SQL, a permis de créer un ver se propageant rapidement sur internet, avec deni de service à la chaîne. CVE (Common Vulnerabilities and Exposure) 2002-0649
- ▶ Heartbleed : débordements en lecture incontrôlés dans tableau du protocole TLS - CVE 2014-0160. Fuite possible des clés privées, mots de passe, etc. sur la moitié des serveurs https de la planète entre 2012 et 2014.
- ▶ Panne des centres d'appels d'urgence en juin 2021, suite à introduction d'un débordement mémoire incontrôlé, après mise à jour logicielle chez ORANGE (5 morts).

Liste non exhaustive !

En fil conducteur : un autre bogue célèbre

Un bogue présent dans la plupart des bibliothèques pendant des décennies... Le voyez-vous ?

```
1  int binary_search(long t[], int n, long v) {
2      int l = 0, u = n-1;
3      while (l <= u) {
4          int m = (u + 1) / 2; ...
5          if (t[m] < v) l = m + 1;
6          else if (t[m] > v) u = m - 1;
7          else return m;
8      }
9      return -1;
10 }
```

Trouvé en 2006, quand Google a commencé à manipuler des tableaux de grandes tailles !

Correction ? ...

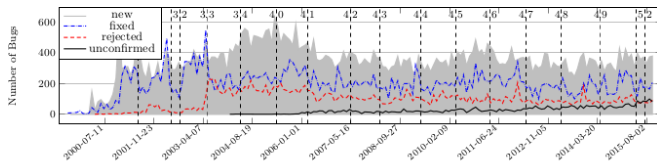
Construire des logiciels sans bogue (non intentionnel)...

Les “bonnes pratiques” du *génie logiciel* :

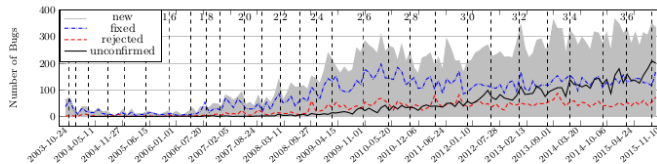
- ▶ Gestion de projet (e.g. scrum, git, intégration continue)
- ▶ Abstractions des langages de prog. (e.g. fonctions, types)
- ▶ Patrons de conception (e.g. modèle-vue-contrôleur en prog. web)
- ▶ Vérifications statiques (e.g. typage)
- ▶ Vérifications dynamiques (e.g. prog. défensive, tests)

Bogues dans GCC et LLVM, deux compilos conformes à l'état de l'art du génie logiciel

cf. “*Toward understanding compiler bugs in GCC and LLVM*” [Sun-et-al@PLDI'16]



(a) GCC.



(b) LLVM

Tandis que des bogues sont régulièrement “fixed”, d'autres améliorations introduisent des “new bugs”.

L'origine des bogues : pensée floue & jugements intuitifs...

1) Cet animal a-t-il de grandes oreilles ?

a. **Non, c'est un canard.**

b. **Oui, c'est un lapin.**

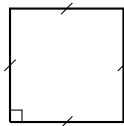


fr.wikipedia.org/wiki/Canard-lapin

2) Est-ce un triangle ou un rectangle ?

a. **Oui.**

b. **Non, c'est un carré.**



3) Mettre au passif la phrase

"Ce prof estime les étudiants brillants".

a. **Les étudiants sont estimés brillants par ce prof.**

b. **Les étudiants brillants sont estimés par ce prof.**

4) Un "café crème" coûte 1,1€. Sachant que le café coûte 1€ de plus que la crème, combien coûte la crème ?

La crème coûte 0,05€ (et le café 1,05€).

Attention aux "heuristiques de jugement" ! (D. Kahneman).

Plan du Cours 1

Les bogues logiciels : conséquences, causes et contre-mesures

Introduction aux méthodes formelles du génie logiciel

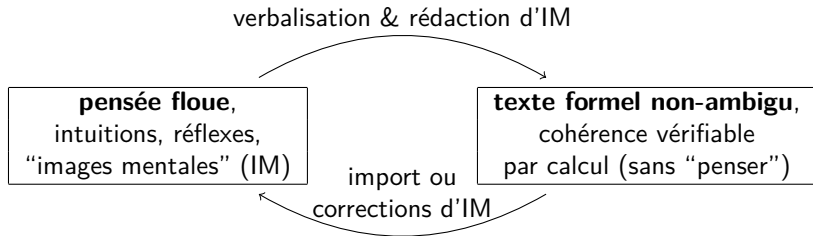
Contrôler la pensée floue par la mathématique

La mathématique est un langage :

- *pour définir des concepts non-flous et non paradoxaux ;*
- *et vérifier que les raisonnements sur ces concepts sont corrects.*

Logique formelle : déf. **mathématique** de ce langage
avec raisonnements vérifiables **mécaniquement** (par ordinateur).

Interactions “intuition” & “formalisation”



Méthodes formelles : mathématiser le logiciel pour circonscrire ses bogues

Idéal réduire la correction du logiciel à une *partie restreinte* la **Trusted Computing Base (TCB)** avec hypothèses & modèles explicites et formalisés de son environnement.

Exemple CompCert (et Chamois CompCert sa variante grenobloise)
Un compilateur C formellement vérifié dans le prouveur Rocq.

La TCB inclut : modèles du langage C, des assembleurs, ABI,...

Prouve : traduction du C en assembleur avec optimisations.

↪ code généré proche de GCC-O1, mais beaucoup plus sûr !

Commercialisé dans chaîne logicielle d'AbsInt pour logiciels critiques embarqués (avionique, nucléaire, automobile, etc).

Qq outils de méthodes formelles (dont † ceux utilisés en ACSS)

Vérification déductive de programmes

Plugin WP[†] de FRAMA-C (preuve de programmes C)

CREUSOT (preuve de programmes RUST)

Preuve de théorèmes

Assistant de preuve ROCQ (preuve interactive)

Alt-Ergo[†] (SMT-solving automatique)

Analyse statique

ASTRÉE et Plugin EVA[†] de FRAMA-C (analyses de prog. C)

GHIDRA[†] (analyse/reverse de binaires)

Exécution concolique (*bug finding*)

KLEE[†] (dans l'IR *middle-end* du compilateur LLVM)

SAGE (outil interne à Microsoft)

Model Checking

SPIN (dev. vérifié de programmes multi-threadés)

JAVA PATHFINDER (détection data-races dans prog. Java)

Preuve d'absence d'UB dans `binary_search` corrigé

avec plugins WP et RTE de FRAMA-C (25.0)

Via “`frama-c-gui -wp -wp-rte signed_binary_search_rte.c`”

Sous **pré-condition** (donnée par `requires` ci-dessous) que tableau `t` bien alloué pour taille `n` fournie.

```
/*@ requires 0 <= n && \valid(t+(0..n-1));  
    @ assigns \nothing;  
    @ ensures -1 <= \result < n;  
    @ ensures \result >= 0 ==> t[\result] == v;*/  
int binary_search(long t[], int n, long v)
```

Prouve au passage 3 **post-conditions** : retourne un entier de $[-1, n[$ qui **lorsque positif** donne indice de valeur `v` recherchée (clauses `ensures`), sans rien modifier pour client (cf. `assigns`).

Preuve qui nécessite un **invariant de boucle** en annotation.

Comment fonctionne cet outil ?

Il commence par insérer les **assertions** RTE (Run-Time Errors) qui garantissent absence de *Undefined Behavior* (UB) ce qui inclut débordement arithmétique et de tableau.

À partir de la **spécification** (clauses requires, assigns et ensures), et des annotations (cf . loop invariant et loop assigns), il génère **obligations de preuves** (OP) pour garantir que le code vérifie les assertions et les postconditions.

Les OP les plus complexes sont déchargées au solveur SMT
ALT-ERGO.

La suite du cours explique quelles OP sont générées et comment...