INF889A

Analyse de programmes pour la sécurité logicielle

Chapitre 2 - Représentation intermédiaire et instrumentation

Quentin Stiévenart

Hiver 2024

Références

- Architecture de LLVM : The Architecture of Open Source Applications (Volume 1), Chap. 11 (un peu daté)
- LLVM IR :
  - Mapping High Level Constructs to LLVM IR
  - LLVM Language Reference Manual
- Passes LLVM : Writing an LLVM Pass
- LLVM For Security
  - 1/4
  - 3/4

Représentation intermédiaire (IR)

Compilation

Un **compilateur** traduit un programme depuis un langage source vers un langage cible

- généralement d'un langage *haut-niveau* vers un langage *bas-niveau* ou vers du code machine

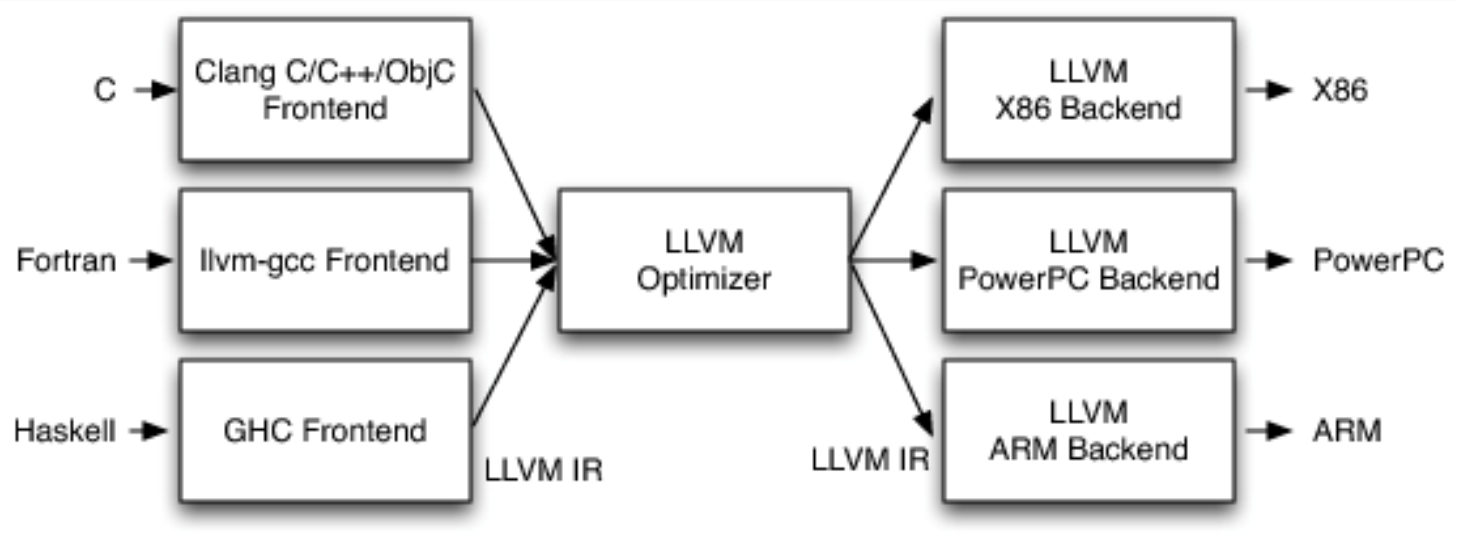
Exemple : C $\rightarrow$ x86_64

Cours pertinents : INF600E, INF7641

Structure d'un compilateur

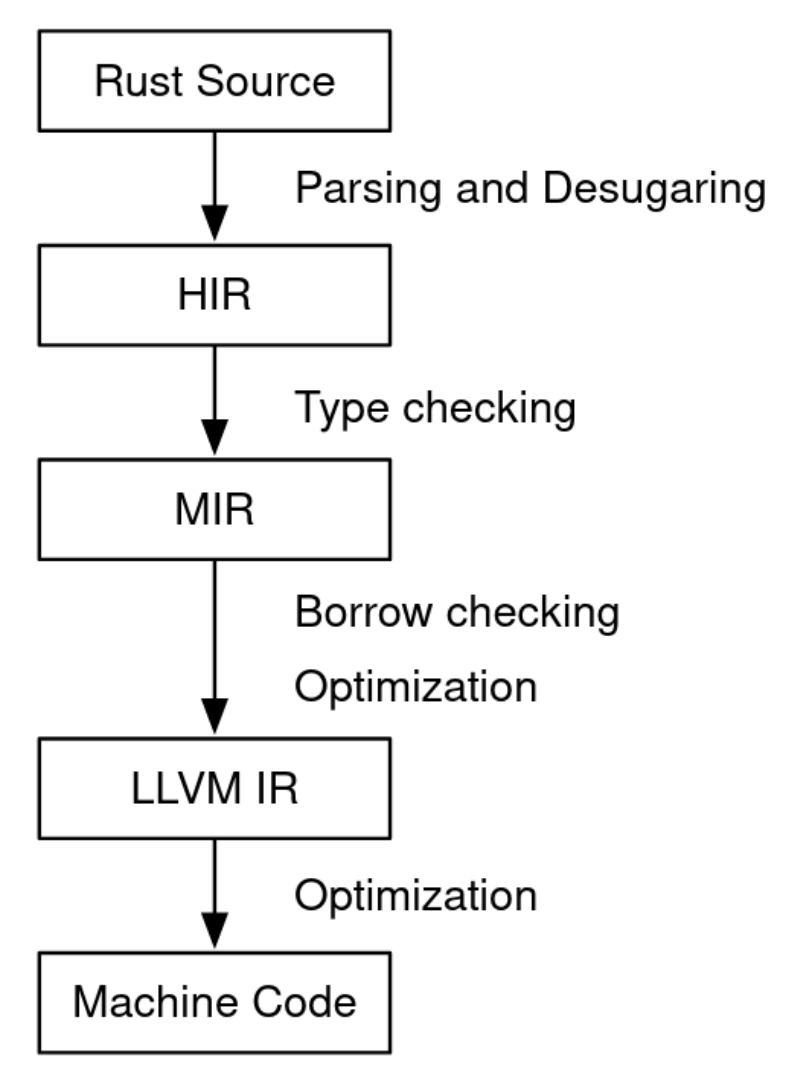
Généralement implémenté en une série de phases :

- Analyse lexicale (*lexer*) : chaîne de caractères $\rightarrow$ tokens
- Analyse syntaxique (*parsing*) : tokens $\rightarrow$ AST / IR
- **Analyse sémantique, optimisations (plusieurs passes): IR $\rightarrow$ IR**
- Génération de code : IR $\rightarrow$ code machine



Structure d'un compilateur

Exemple : compilateur Rust (source)



Analyse sémantique

Pourquoi s'intéresser à l'analyse sémantique d'un compilateur ?

- langage plus simple que le langage source
- peu de perte d'information contrairement au langage cible
- niveau idéal pour l'instrumentation
- adapté à l'analyse statique

LLVM

> *The LLVM Project is a collection of modular and reusable compiler and toolchain technologies* – *llvm.org*

Construit en opposition aux approches monolithiques de compilation, tel que fait par GCC.

Collection de projets :

- LLVM Core : bibliothèques, indépendantes du langage source et destination
- Clang : front-end pour compiler du C, C++, Objective-C
- LLDB : debugger
- LLD : linker
- ...

LLVM IR

> *LLVM IR is designed to host mid-level analyses and transformations that you find in the optimizer section of a compiler. It was designed with many specific goals in mind, including supporting lightweight runtime optimizations, cross-function/interprocedural optimizations, whole program analysis, and aggressive restructuring transformations, etc. The most important aspect of it, though, is that it is itself defined as a first class language with well-defined semantics.*

The Architecture of Open Source Applications, Volume 1

LLVM IR : Aperçu

code/hello.c

```c
#include <stdio.h>

int main() {
  printf("Hello, world\n");
  return 0;
}
```

À compiler en LLVM IR avec :

```
$ clang -S -emit-llvm hello.c
```

Image Docker avec LLVM

```
$ docker run -v /chemin/vers/code/:/code -it \
    registry.gitlab.info.uqam.ca/inf889a/slides/llvm
# export PATH=/llvm-project/build/bin/:$PATH
# clang -S -emit-llvm /code/hello.c
```

LLVM IR : Aperçu

code/hello.ll

```
; ModuleID = 'hello.c'
source_filename = "hello.c"
target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:
target triple = "x86_64-pc-linux-gnu"

@.str = private unnamed_addr constant [14 x i8] c"Hello, world\0

; Function Attrs: noinline nounwind optnone sspstrong uwtable
define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4
  store i32 0, ptr %1, align 4
  %2 = call i32 (ptr, ...) @printf(ptr noundef @.str)
  ret i32 0
}
```

Caractéristiques de LLVM IR

Une instruction est composée de :

- une mnémonique
- un nombre d'entrées
- un résultat
- de l'information de typage

```
%c = add i32 %a, %b
```

L'IR est indépendante de la machine :

- nombre infini de « registres » : `%0`, `%1`, ...
- convention d'appel : `call` et `ret`

LLVM IR : Formats

Plusieurs formats équivalents :

- format textuel : fichiers `.ll`
- structure de données manipulée par les outils
- format binaire : fichiers `.bc`

On peut passer de `.ll` à `.bc` avec `llvm-as` et `llvm-dis`

Design de LLVM IR

Design de LLVM IR

The intermediate representation of a compiler is interesting because it can be a "perfect world" for the compiler optimizer: unlike the front end and back end of the compiler, the optimizer isn't constrained by either a specific source language or a specific target machine. On the other hand, it has to serve both well: it has to be designed to be easy for a front end to generate and be expressive enough to allow important optimizations to be performed for real targets.

SSA

LLVM IR utilise la *forme SSA*

- *static single assignment form*
- chaque variable ne peut être assignée qu'une seule fois
- on a une information explicite sur la provenance des valeurs

Habituellement :

```
y = 1
y = 2
x = y
```

En forme SSA :

```
y1 = 1
y2 = 2
x1 = y2
```

Référence du langage

Pour tous les détails, voir la référence du langage

Pour une introduction pratique, voir Mapping High Level Constructs to LLVM IR

LLVM IR : types de données

Entiers : `iN` où `N` est la taille :

- `i1` : booléen
- `i32`

Pointeurs :

- `ptr` (LLVM ≥ 15)
- `i32*`

LLVM IR : variables globales

Préfixées avec un `@`, traitées comme des pointeurs (e.g., `i32*`)

```
@variable = global i32 21 ; initialisée à 21

define i32 @main() {
    %1 = load i32, i32* @variable ; chargement de la variable
    %2 = mul i32 %1, 2            ; multiplication par 2
    store i32 %2, i32* @variable  ; modification de la variable
    ret i32 %2                    ; valeur de retour
}
```

LLVM IR : registres

Attention à la forme SSA :

```
%tmp = add i32 4, 2
%tmp = add i32 4, 1  ; Incorrect! %tmp a déjà été écrit une fois
```

Chaque registre doit être définit de façon unique :

```
%tmp.0 = add i32 4, 2
%tmp.1 = add i32 4, 1
```

LLVM IR : variables locales

Allouées sur la pile avec `alloca`, qui retourne un pointeur

```
int a = 0;
a = a + 1;
```

devient :

```
%a = alloca i32                ; déclare a
store i32 0, i32* %a           ; initialise a

%a.1 = load i32, i32* %a ; charge la valeur de a
%a.2 = add i32 %a.1, 1   ; incrémente la valeur
store i32 %a.2, i32 %a   ; écrit la nouvelle valeur
```

LLVM IR : branchements

Possibilité de nommer des blocs avec des labels, et de sauter avec `br`

```
int max(int a, int b) {
  if (a > b) {
    return a;
  } else {
    return b;
  }
}
```

LLVM IR : branchements

```
define i32 @max(i32 %a, i32 %b) {
entry:
  %retval = alloca i32
  %0 = icmp sgt i32 %a, %b
  br i1 %0, label %btrue, label %bfalse
btrue:
  store i32 %a, i32* %retval
  br label %end
bfalse:
  store i32 %b, i32* %retval
  br label %end
end:
  %1 = load i32, i32* %retval
  ret i32 %1
}
```

LLVM IR : nœuds ϕ

On aimerait définir ceci, mais ce n'est pas valide : `%retval` est défini deux fois

```
define i32 @max(i32 %a, i32 %b) {
entry:
  %0 = icmp sgt i32 %a, %b
  br i1 %0, label %btrue, label %bfalse
btrue:
  %retval = %a
  br label %end
bfalse:
  %retval = %b
  br label %end
end:
  ret i32 %retval
}
```

LLVM IR : nœuds ϕ

Solution : l'instruction `phi` qui sélectionne une valeur en fonction d'où on vient

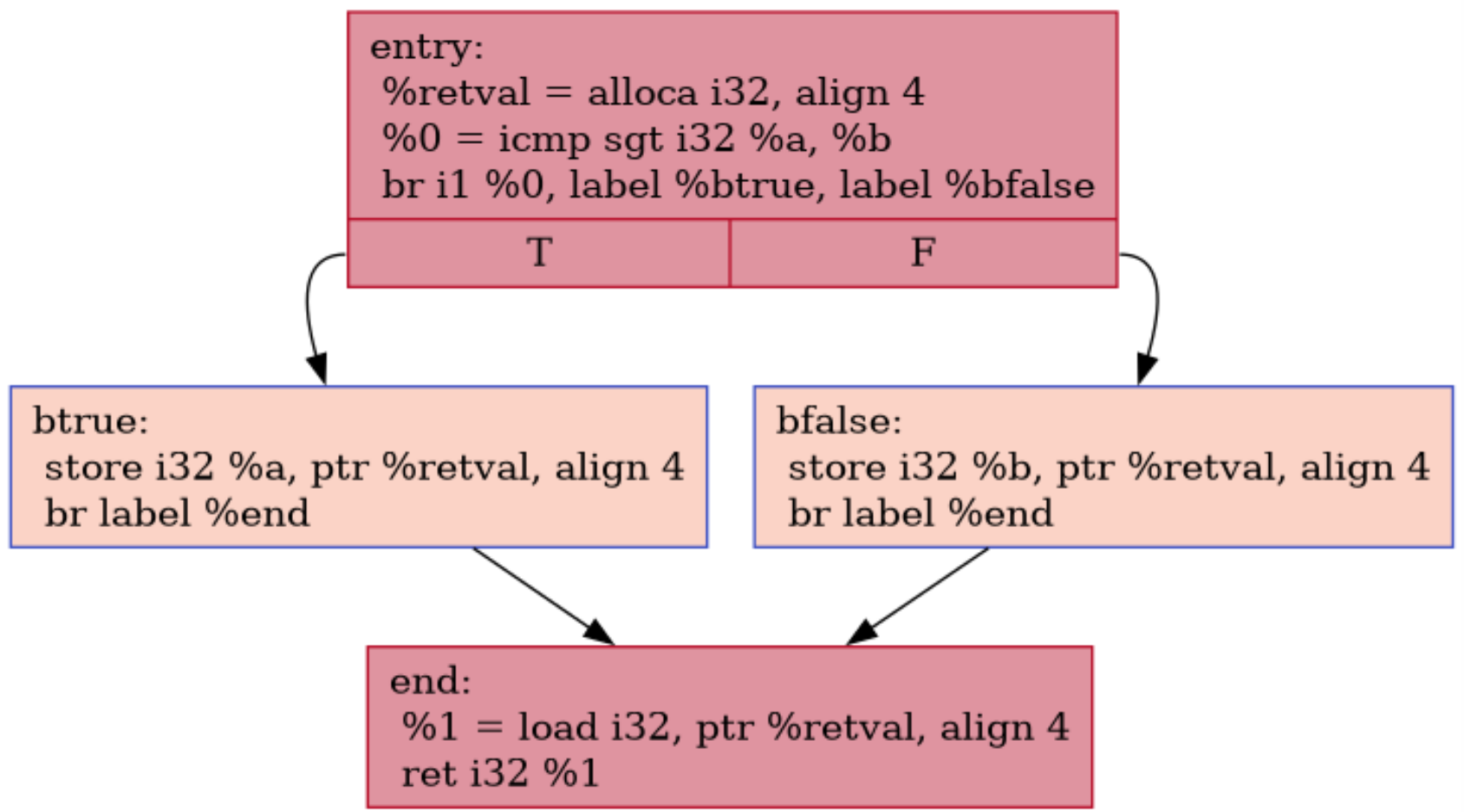
```
define i32 @max(i32 %a, i32 %b) {
entry:
  %0 = icmp sgt i32 %a, %b
  br i1 %0, label %btrue, label %bfalse
btrue:
  br label %end
bfalse:
  br label %end
end:
  %retval = phi i32 [%a, %btrue], [%b, %bfalse]
  ret i32 %retval
}
```

Visualisation d'un programme

On peut visualiser le flot de contrôle d'un programme au travers de son *control-flow graph* (CFG)

```
$ opt -disable-output -passes=dot-cfg code/max.ll
$ xdot .max.dot
```

Visualisation d'un programme



CFG for 'max' function

LLVM IR : Exercices

Traduire le programme suivant en LLVM IR en deux versions :

- une qui représente les variables locales par des registres
- une qui représente les variables locales par des variables sur la pile

```
int foo(int input) {
  int a, b, c;
  a = 42;
  b = 87;
  if (input)
    c = a + b;
  else
    c = a - b;
  return c;
}
```

LLVM IR : Exercice

Comparer votre version avec le code généré par LLVM (`-S -emit-llvm`).

Passes LLVM

LLVM effectue les tâches d'analyse, instrumentation, et d'optimisation au travers de *passes*

L'outil `opt` permet de sélectionner des passes à faire tourner

Propagation de constante

code/constprop.ll

```
define i32 @constprop() {
  %1 = add i32 1, 2
  %2 = icmp sgt i32 %1, 0

  br i1 %2, label %btrue, label %end

btrue:
  br label %end

end:
  ret i32 1
}
```

Propagation de constante

code/constprop.ll

```
$ opt -passes=sccp constprop.ll -S

define i32 @constprop() {
  br label %btrue

btrue:
  br label %end

end:
  ret i32 1
}
```

Une autre passe d'optimisation pourra enlever les sauts inutiles

Passes LLVM en pratique

Voir toutes les passes exécutées :

```
$ opt -O2 -print-after-all -S -disable-output hello.ll \
    2>&1 | grep Dump
```

Voir toutes les passe disponibles :

```
$ opt -print-passes
```

Programmer avec LLVM

Programmer avec LLVM

LLVM est une grosse base de code : plus de 7 millions de lignes de C ou C++ :

- les outils intégrés à votre éditeur peuvent avoir des problèmes vu la taille du projet
  - utiliser `find` + `grep`
  - ou `ctags`
- on va se limiter à quelques éléments dans `llvm/lib` et `llvm/include`
- la doc doxygen peut être utile
- les commentaires aussi
- c'est du C++: il y a du code dans les *headers* et dans les `.cpp`

Implémenter une passe LLVM

Manager de passe

LLVM : Using the New Pass Manager

> *LLVM currently contains two pass managers, the legacy PM and the new PM. The optimization pipeline (aka the middle-end) uses the new PM, whereas the backend target-dependent code generation uses the legacy PM.*
>
> *Some IR passes are considered part of the backend codegen pipeline even if they are LLVM IR passes*

Simplification d'instruction

code/Simplify/

On souhaite simplifier certaines instructions :

- `%y = sub i32 %x, 0`: on peut éliminer `%y` et le remplacer par `%x`
- `%x = add i32 a, b` où `a` et `b` sont des constantes : on peut éliminer `%x` et le remplacer par `a+b`

Structure d'un plugin de passe

`code/Simplify/`

Au minimum :

- Un `CMakeLists.txt` pour compiler
- Le fichier source de la passe

Compiler la passe

code/Simplify

```
$ export LLVM_DIR=/chemin/vers/llvm-project/build/
$ mkdir build && cd build
$ cmake -DLT_LLVM_INSTALL_DIR=$LLVM_DIR .. && make
[ 50%] Building CXX object CMakeFiles/Simplify.dir/Simplify
[100%] Linking CXX shared library libSimplify.so
[100%] Built target Simplify
```

Lancer la passe

code/Simplify

```
$ $LLVM_DIR/bin/opt -load-pass-plugin=libSimplify.so \
    -passes=simplify ../example.ll -S
```

Avec `example.ll` :

```
define i32 @foo(i32 %a, i32 %b) {
  %c = add i32 1, 2
  %d = sub i32 %a, 0
  %e = add i32 %a, %b
  %f = add i32 %c, %d
  %g = add i32 %e, %f
  ret i32 %g
}
```

Résultat de la passe

code/Simplify

```
define i32 @foo(i32 %a, i32 %b) {
  %e = add i32 %a, %b
  %f = add i32 3, %a
  %g = add i32 %e, %f
  ret i32 %g
}
```

Instrumentation

Instrumentation

L'instrumentation modifie le code (source ou objet) d'un programme afin d'altérer ou d'observer son comportement.

Souvent utilisé pour :

- détecter des incohérence lors de l'exécution d'un programme : accès hors des bornes d'un tableau, accès mémoire invalide, problèmes de concurrence
- extraire des traces d'exécution
- calculer la couverture

L'instrumentation a un impact sur le temps d'exécution.

Exemple : BoundsChecking

code/bounds.c

```c
#include <stdio.h>

int main() {
  int arr[16];
  for (int i = 0; i < 16; i++) {
    arr[i] = i;
  }
  return 0;
}
```

Exemple : `BoundsChecking` `code/bounds.c`

Compiler sans aucune passe :

```
$ clang -S -emit-llvm -fno-discard-value-names bounds.c
```

- `-S -emit-llvm` produit du code LLVM IR
- `-fno-discard-value-names` préserve les noms de variables pour les registres LLVM

Exemple : `BoundsChecking` — `code/bounds.c`

Lancer uniquement la passe `bounds-checking` :

```
$ opt -passes=bounds-checking -S -o bounds.after.ll bounds.ll
```

Observer les ajouts :

```
$ diff --side-by-side bounds.ll bounds.after.ll
```

Exemple : BoundsChecking

code/bounds.c

Ce qui a été rajouté :

```
  %arrayidx.idx = mul i64 %idxprom, 4
  %3 = add i64 0, %arrayidx.idx      ; %3 = i*4
  %4 = sub i64 64, %3                ; %4 = 64-%3, 64 = taille du tablea
  %5 = icmp ult i64 64, %3           ; 64 < i*4, donc est-ce qu'on
                                     ; indexe après la fin du tableau
  %6 = icmp ult i64 %4, 4            ; %4 < 4, donc est-ce que le tablea
                                     ; a moins d'un élément
  %7 = or i1 %5, %6                  ; une condition est-elle vraie ?
  br i1 %7, label %trap, label %8    ; si oui, on est hors des bornes
...
trap:
 call void @llvm.trap() #3
 unreachable
```

Trouver le code de `BoundsChecking`

Pour trouver le code de cette passe :

```
$ cd llvm-project/llvm
$ rg bounds-checking lib/
lib/Passes/PassRegistry.def
272:FUNCTION_PASS("bounds-checking", BoundsCheckingPass())
$ rg BoundsChecking
include/llvm/Transforms/Instrumentation/BoundsChecking.h
19:struct BoundsCheckingPass : PassInfoMixin<BoundsChecking
$ find . -name BoundsChecking\*
./lib/Transforms/Instrumentation/BoundsChecking.cpp
./test/Instrumentation/BoundsChecking
./include/llvm/Transforms/Instrumentation/BoundsChecking.h
```

Implémentation de `BoundsChecking` `LLVM:BoundsChecking`

Ordre conseillé de lecture :

- `BoundsChecking.h` : uniquement des déclarations ici (pas toujours le cas)
- `BoundsChecking.cpp` :
  - `run`
  - `addBoundsChecking`
  - `getBoundsCheckCond`
  - `insertBoundsCheck`

Implémentation de `BoundsChecking` `LLVM:BoundsChecking`

`run` :

- `AM.getResult` permet d'obtenir le résultat d'autres analyses
- `addBoundsChecking` va ajouter les checks, et retourner vrai si le code a changé
- `PreservedAnalyses` permet de savoir s'il faut invalider le résultat d'autres analyses
  - Si on a changé le code, on invalide tout

Implémentation de `BoundsChecking` `LLVM:BoundsChecking`

`addBoundsChecking` procède en deux étapes :

```
for (Instruction &I : instructions(F)) {
```

- On repère les instructions à protéger
  - `load`
  - `store`
  - `cmpchg` et `atomicrmw` (modifications atomiques)
- `getBoundsCheckCond` nous donne le `or` qui vérifie si on est hors borne

```
for (const auto &Entry : TrapInfo) {
```

- On insère les checks en utilisant `insertBoundsCheck`

Implémentation de `BoundsChecking` `LLVM:BoundsChecking`

`getBoundsCheck`

Construit la condition et la retourne

```
// three checks are required to ensure safety:
// . Offset >= 0  (since the offset is given from the bas
// . Size >= Offset  (unsigned)
// . Size - Offset >= NeededSize  (unsigned)
//
// optimization: if Size >= 0 (signed), skip 1st check
```

Implémentation de `BoundsChecking` `LLVM:BoundsChecking`

`insertBoundsCheck`

Ajoute l'instruction `br`, adapte la structure de bloc

Note : une instruction `br` doit être la dernière instruction d'un bloc

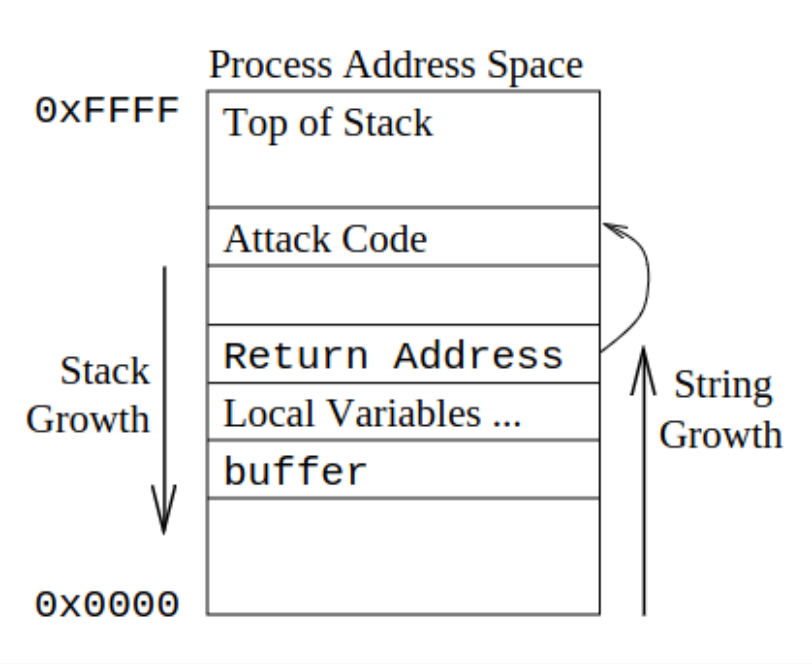
Cowan. et al. (1998). Stackguard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. USENIX.

Implémenté dans LLVM dans `lib/CodeGen/StackProtector.cpp`

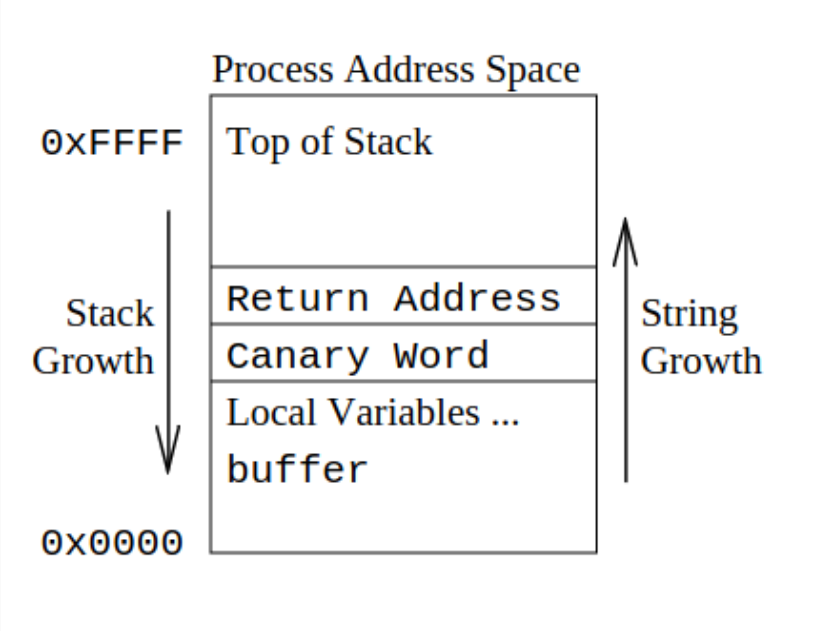
- Fait a la génération car l'implémentation dépend de l'architecture

StackGuard : idée

Cowan et al., USENIX 1998



Sans protection



Avec `StackGuard`

StackGuard : Code C

code/stack.c

```c
#include <unistd.h>

int main() {
  int x = 0;
  int buf[16];
  read(0, buf, 32);
  return x;
}
```

StackGuard : sans et avec

code/stack.c

```
$ clang -S -fno-stack-protector stack.c -o stack.without.s
$ clang -S -fstack-protector stack.c -o stack.with.s
$ diff --side-by-side stack.without.s stack.with.s
```

Sauve la valeur du canary (mise dans `%fs:40` par glibc) dans `rbp-8` :

```
movq    %fs:40, %rax
movq    %rax, -8(%rbp)
```

Récupère les même valeurs avant la fin de la fonction :

```
movq    %fs:40, %rax
movq    -8(%rbp), %rcx
```

Rate si ce ne sont pas les mêmes :

```
  cmpq    %rcx, %rax
  jne     .LBB0_2
  ...
.LBB0_2:
  .cfi_def_cfa %rbp, 16
  callq   __stack_chk_fail@PLT
```

StackGuard : implémentation

Dans `CodeGen/StackProtector.cpp`

En simplifié :

```
bool StackProtector::runOnFunction(Function &Fn) {
  if (!RequiresStackProtector())
    return false;

  bool Changed = InsertStackProtectors();
  return Changed;
}
```

Même si elle est située dans le CodeGen, c'est une passe comme une autre.

Exemple: AddressSanitizer

Serebryany, Bruening, Potapenko, & Vyukov. (2012). AddressSanitizer: A Fast Address Sanity Checker. Usenix Security.

Implémenté dans LLVM :

```
$ wc -l ./llvm/lib/Transforms/Instrumentation/AddressSanitizer.c
3499 ./llvm/lib/Transforms/Instrumentation/AddressSanitizer.cpp
```

AddressSanitizer : but et résultats — Serebryany et al., 2012

AddressSanitizer vise à :

- détecter les erreurs mémoires
  - accès hors-borne dans le tas, la pile, les objets globaux
  - *use-after-free*
- de façon efficace
  - Les approches existantes sont trop lentes ou limitées pour être utilisées en pratique (exemple: Valgrind $\times 10$ à $\times 20$)

Résultats :

- ralentissement de 73% en moyenne
- 300 bugs trouvés dans Chromium lors de la publication

Approche similaire à Valgrind, mais avec une attention à la rapidité :

- **Shadow Memory** : on a une « copie » de la mémoire qui nous dit quelles zones sont sûres à l'accès
- Instrumentation du programme pour vérifier que les accès sont sûrs (`load` et `store`)
- Une *runtime library* qui gère la *shadow memory*
- Une gestion particulière de la stack et des globales

AddressSanitizer : *shadow memory* Serebryany et al., 2012

Observation : `malloc` retourne des adresses alignées sur 8 octets

- Une séquence de 8 octets alignée peut être dans 9 états différents :
  - tout est adressable
  - rien n'est adressable
  - les k premiers octets sont adressables Par exemple, `malloc(13)` nous donne 8+5 octets adressables

$\rightarrow \frac{1}{8}$ se l'espace mémoire est dédié à *shadow memory*

Règle de conversion : `ShadowAddr = (Addr >> 3) + Offset`

- `Addr` est l'adresse "réelle"
- `ShadowAddr` est l'adresse équivalente en shadow memory

AddressSanitizer : *shadow memory* Serebryany et al., 2012

Valeurs dans la *shadow memory* :

- 0: les 8 octets suivants sont adressables
- k: les k prochains octets sont adressables, les 8-k restant ne le sont pas
- valeur négative : aucun octet parmi les 8 suivant n'est adressable

AddressSanitizer : instrumentation Serebryany et al., 2012

Un accès mémoire de 8 octets est remplacé par :

```
ShadowAddr = (Addr >> 3) + Offset;
if (*ShadowAddr != 0)
  ReportAndCrash(Addr);
```

Si accès à 1, 2, ou 4 octets seulement :

```
ShadowAddr = (Addr >> 3) + Offset;
k = *ShadowAddr;
if (k != 0 && ((Addr & 7) + AccessSize > k)
  ReportAndCrash(Addr);
```

AddressSanitizer: Instrumentation Serebryany et al., 2012

We placed the AddressSanitizer instrumentation pass at the very end of the LLVM optimization pipeline. This way we instrument only those memory accesses that survived all scalar and loop optimizations performed by the LLVM optimizer. For example, memory accesses to local stack objects that are optimized away by LLVM will not be instrumented. At the same time we don't have to instrument memory accesses generated by the LLVM code generator (e.g., register spills)

AddressSanitizer : *Runtime library* Serebryany et al., 2012

Cette bibliothèque est chargée de gérer la *shadow memory*

- À l'initialisation du programme
  - `mmap` la *shadow memory* dans son entièreté
  - protège les adresses correspondant à la *shadow memory* dans elle-même
- Remplace `malloc` :
  - Alloue une *redzone* autour de la région allouée, marquée non-adressable
  - La *redzone* contient des méta-données
  - En cas d'overflow, les *redzone* seront écrasées
- Remplace `free` :
  - Désalloue la mémoire et la marque non-adressable dans son entièreté

AddressSanitizer : pile et globales — Serebryany et al., 2012

Idée : ajout de *redzones* autour de chaque variable globale ou sur la pile :

```
void foo() {
  char rz1[32];
  char arr[10]; // variable locale
  char rz3[32-10+32];
  unsigned *shadow = (unsigned*)(((long)rz1>>8)+Offset);
  shadow[0] = 0xffffffff; // rz1 est non-adressable
  shadow[1] = 0xffff0200; // arr est adressable, pas rz2
  shadow[2] = 0xffffffff; // rz2 est non-adressable
  ... // code de la fonction
  // rerend tout adressable
  shadow[0] = shadow[1] = shadow[2] = 0;
}
```

AddressSanitizer : faux négatifs — Serebryany et al., 2012

Faux négatif = un accès non valide n'est pas détecté

Accès à de la mémoire non-alignée partiellement hors-borne :

```
int *a = new int[2]; // aligné sur 8 octet
int *u = (int*)((char* a) + 6);
*u = 1; // Accède les octets [6, 9]
```

Accès distant à de la mémoire accessible :

```
char *a = new char[100];
char *b = new char[1000];
a[500] = 0; // écrit dans b, malgré la redzone entre a et b
```

C'est un choix de ne pas considérer ces cas, dans le but de rester efficace.

AddressSanitizer : faux positifs — Serebryany et al., 2012

Faux positif = un accès valide est reporté comme invalide

> *In short, AddressSanitizer has no false positives.*

À une condition : ajouter `no_address_safety_analysis` pour du code très particulier (rencontré une fois dans Chromium)

> *For example, low-level code iterates between two addresses on the stack crossing multiple stack frames*

Serebryany, & Iskhodzhanov. (2009). ThreadSanitizer: Data Race Detection in Practice, WBIA.

Instrumentation :

```
// The instrumentation phase is quite simple:
//   - Insert calls to run-time library before every memory
//      - Optimizations may apply to avoid instrumenting so
//   - Insert calls at function entry/exit.
// The rest is handled by the run-time library.
```

Exemple: ThreadSanitizer

Serebryany et al., 2009

Run-time :

- à chaque accès, calcule une représentation de l'état et des relations *happens-before*
- si l'état correspond à une *data race*, c'est reporté

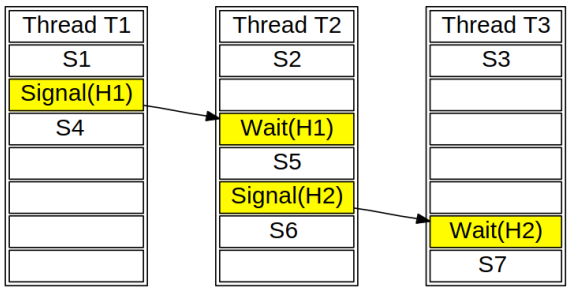


Figure 1: Example of happens-before relation. $S_1 \prec S_4$ (same thread); $S_1 \prec S_5$ (happens-before arc $\text{SIGNAL}_{T_1}(H_1) - \text{WAIT}_{T_2}(H_1)$); $S_1 \prec S_7$ (happens-before is transitive); $S_4 \not\prec S_2$ (no relation).

Autres utilisations de l'instrumentation dans LLVM

Sanitizers :

- `DataFlowSanitizer` : framework générique pour implémenter d'autres sanitizers (doc)
- `HWAddressSanitizer` : AddressSanitizer en matériel (AArch64 et Intel LAM)
- `KCFI` : protège les appels indirects
- `MemorySanitizer` : détecte les lectures dans de la mémoire non initialisée (slowdown: 3×)

Autres utilisations de l'instrumentation dans LLVM

Profilage et couverture :

- `SanitizerCoverage` : calcul de couverture paramétrisable, utile pour des fuzzers
- `GCOVProfiler` : pour profiler avec `gcov`
- `InstrProfiling` et `PGOInstrumentation` : utilisé pour faire du *profile-guided optimization*.
- `MemProfiler` : profilage mémoire

Liens avec des approches d'analyse

La majorité des approches dynamiques ont besoin d'une forme d'instrumentation

Fuzzing (chap. 1) : on veut généralement utiliser une forme d'instrumentation

- pour mesurer la couverture
- pour détecter des états d'erreurs

Exécution symbolique dynamique (chap. 3) :

- pour enregistrer les branchements et chemins d'exécutions

Instrumentation : articles

Sanitizers :

- Duck & Yap. (2018). EffectiveSan: Type and Memory Error Detection using Dynamically Typed C/C++, PLDI.
- Haller et al. (2016). TypeSan: Practical Type Confusion Detection, CCS.
- van der Krouwe et al. (2017). DangSan: Scalable Use-after-free Detection, EuroSys.
- Duck et al. (2017). Stack Bounds Protection with Low Fat Pointers, NDSS.
- Kuvaiskii et al. (2017). SGXBOUNDS: Memory Safety for Shielded Execution, EuroSys.

Instrumentation : articles

Sécurité :

- Polino et al. (2017). Measuring and Defeating Anti-Instrumentation-Equipped Malware, DIMVA.
- Davi et al. (2011). ROPdefender: A Detection Tool to Defend Against Return-Oriented Programming Attacks. AsiaCCS.
- Cai et al. (2019). DroidCat: Effective Android Malware Detection and Categorization via App-Level Profiling, Transactions of Information Forensics and Security.
- Li et al. (2018). K-Hunt: Pinpointing Insecure Cryptographic Keys from Execution Traces, CCS.

Outils d'instrumentation

Instrumentation de code source

- Jalangi : instrumentation de code JavaScript
- Aran : instrumentation de code JavaScript (maintenu)
- `java.instrument`
- LLVM: `DataFlowSanitizer` et `SanitizerCoverage`

Outils d'instrumentation

Instrumentation binaire

Plusieurs framework d'instrumentation binaire :

- DynamoRIO
- Intel Pin
- Valgrind
- Frida

Instrumentation système

- DTrace
- LTTng

Pratique

- Implémenter une passe d'instrumentation pour tracer les appels de fonction, dans LLVM ou avec un outil au choix
- Implémenter LLVM For Security 4/4