Rétro-ingénierie statique

5d54626cc43ba7ce625903ffa39c049c

Groupe de sécurité de l'information, École normale supérieure

2 octobre 2019

TODO

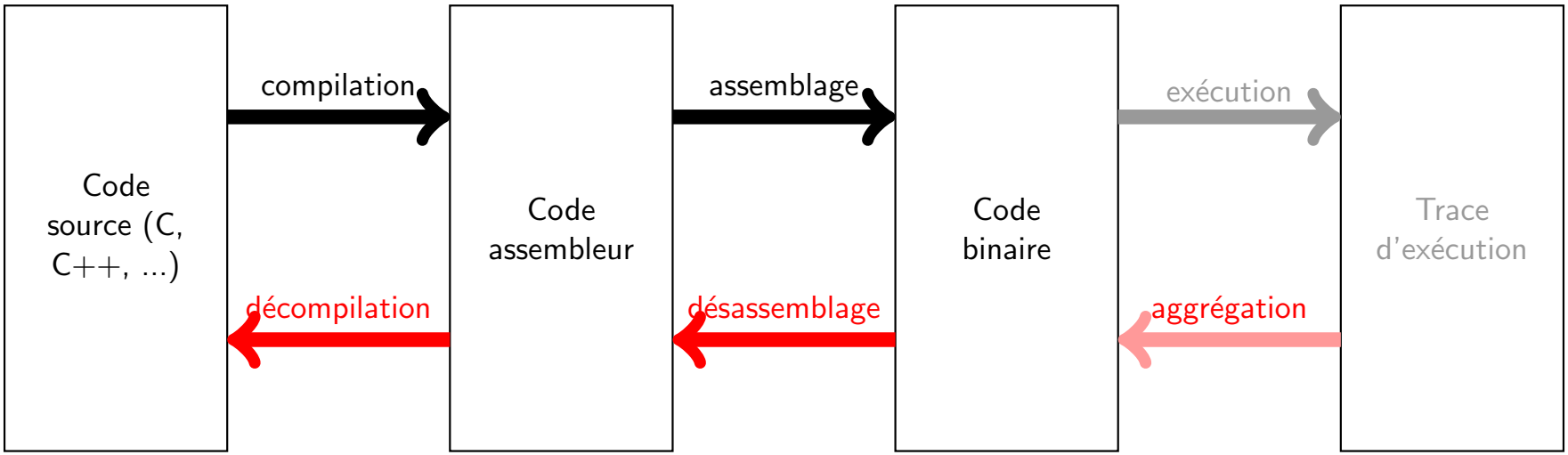
Pour ce cours, merci d'avoir sur votre machine :

- objdump/gcc/readelf
- Ghidra 9.1

Introduction

Definition (Rétro-ingénierie, n.f.)

Ensemble des opérations d'analyse d'un logiciel ou d'un matériel destinées à retrouver le processus de sa conception et de sa fabrication, ainsi que les modalités de son fonctionnement.



Désassemblage

- Le désassemblage repose sur la possibilité de distinguer ce qui est du code et ce qui est de la donnée. Ce problème est **indécidable**.

Proof.

Supposons qu'il existe un programme P qui prend en entrée un flux binaire b, une adresse a, et renvoie un booléen indiquant si les octets commençants à l'octet a sont exécutés. Construisons un programme P' qui prend en entrée un programme X and et une entrée i.

```
1 P'(X,i):
2    T():
3         push i
4         call X
5         ret
6    return P(&T,5)
```

Alors P' est un programme capable de résoudre le problème de l'arrêt. *QED*

Désassemblage

- Il est cependant possible de procéder à des approximations en perdant soit la **complétude**, soit la **correction**.
- De nombreux algorithmes de désassemblage existent.
  - Désassemblage linéaire (linear sweep) : objdump, winDBG
  - Désassemblage récursif : IDA, OllyDBG
  - Autres : Galileo, ...

Linear sweep disassembly

Principe général d'un désassembleur linéaire.

1. Commencer au début de la section de code (`.text`).
2. Désassembler une instruction.
3. Avancer de la longueur de l'instruction
4. Refaire 2-3, tant que l'instruction désassemblée est valide.

Quelques heuristiques lorsqu'une instruction invalide est rencontrée : avancer d'un octet, arrêter, réaligner sur 2, 4, 8 octets...

Linear sweep disassembly

Principe général d'un désassembleur linéaire.

1. Commencer au début de la section de code (`.text`).
2. Désassembler une instruction.
3. Avancer de la longueur de l'instruction
4. Refaire 2-3, tant que l'instruction désassemblée est valide.

Quelques heuristiques lorsqu'une instruction invalide est rencontrée : avancer d'un octet, arrêter, réaligner sur 2, 4, 8 octets...

Angle mort

Incapable de deviner les données incluses dans le code.

Application : objdump

Demo/TP.

Recursive disassembly

1. Commencer à l'adresse d'entrée.
2. Désassembler une instruction.
3. Ajouter à une structure de données, les adresses des instructions suivantes.
4. Faire 2-3 tant que la structure de données n'est pas vide.

Quelques heuristiques : si une instruction invalide est trouvée, si un saut indirect est présent, tables de saut, ...

Recursive disassembly

Pour trouver l'instruction suivante, il est déjà nécessaire de comprendre le type d'instruction que l'on a :

- Instruction séquentielle : l'instruction suivante se trouve immédiatement après.
- Branchement direct conditionnel : les instructions suivantes se trouvent immédiatement après et à l'offset du saut.
- Branchement direct inconditionnel : l'instruction suivante se trouve à l'offset du saut.
- Branchement indirect conditionnel : l'instruction suivante est juste après, l'autre n'est pas accessible sans travail supplémentaire.
- Branchement indirect inconditionnel : aucune information disponible sur l'instruction suivante.

Application : ghidra

Demo/TP.

Réassemblage

Souvent, le réassemblage d'un code désassemblé est impossible :

- Code auto-modifiant.
- Offsets claqués en dur (tables de saut, ...).
- Tables de saut/imports/...
- Optimisations

Nous verrons plus tard comment néanmoins modifier un programme binaire existant.

Galileo

Algorithme de désassemblage utilisé dans le cadre du Return-Oriented Programming (ROP).

Algorithm GALILEO:
create a node, *root*, representing the ret instruction;
place *root* in the trie;
for *pos* **from** 1 **to** *textseg_len* **do**:
if the byte at *pos* is c3, i.e., a ret instruction, **then**:
call BUILDFROM(*pos*, *root*).

Procedure BUILDFROM(index *pos*, instruction *parent_insn*):
for *step* **from** 1 **to** *max_insn_len* **do**:
if bytes $[(pos - step) \ldots (pos - 1)]$ decode as a valid instruction *insn* **then**:
ensure *insn* is in the trie as a child of *parent_insn*;
if *insn* isn't boring **then**:
call BUILDFROM($pos - step$, *insn*).

```
└· 23 [1] -> push rdi
  ├· 20 [3] -> shr edi, 0x28
  ├· 19 [4] -> shr rdi, 0x28
  │ ├· 17 [2] -> jnbe 0x0000000000000066
  │ └·  9 [10] -> mov rdi, 0x647773FFFFFFFFFF
  │   └·  7 [2] -> add al, 0x02
  │     └·  6 [1] -> pop rdx
  │       ├·  5 [1] -> push rax
  │       │ └·  4 [1] -> pop rsi
  │       │   └·  3 [1] -> push rax
  │       │     ├·  1 [2] -> xor eax, eax
  │       │     └·  0 [3] -> xor rax, rax
  │       └·  2 [4] -> rcl byte ptr [rax+0x5E], 0x50
  └· 18 [5] -> shr rdi, 0x28
    ├· 16 [2] -> jnb 0x0000000000000079
    └· 15 [3] -> push [rbx+0x77]
      └· 10 [5] -> mov edi, 0xFFFFFFFF
```

```
 0:   48 31 c0                xor    %rax,%rax
 3:   50                      push   %rax
 4:   5e                      pop    %rsi
 5:   50                      push   %rax
 6:   5a                      pop    %rdx
 7:   04 02                   add    $0x2,%al
 9:   48 bf ff ff ff ff ff    movabs $0x647773ffffffffff,%rdi
10:   73 77 64
13:   48 c1 ef 28            shr    $0x28,%rdi
17:   57                     push   %rdi
```

Galileo : Angle mort

```
mysupergadget:
    do_stuff
    jmp label
label:
    ret
```

Algorithme récursif complet (mais incorrect)

Input: $B_0, ...B_n$, a binary program
Result: G, a directed graph of all execution paths

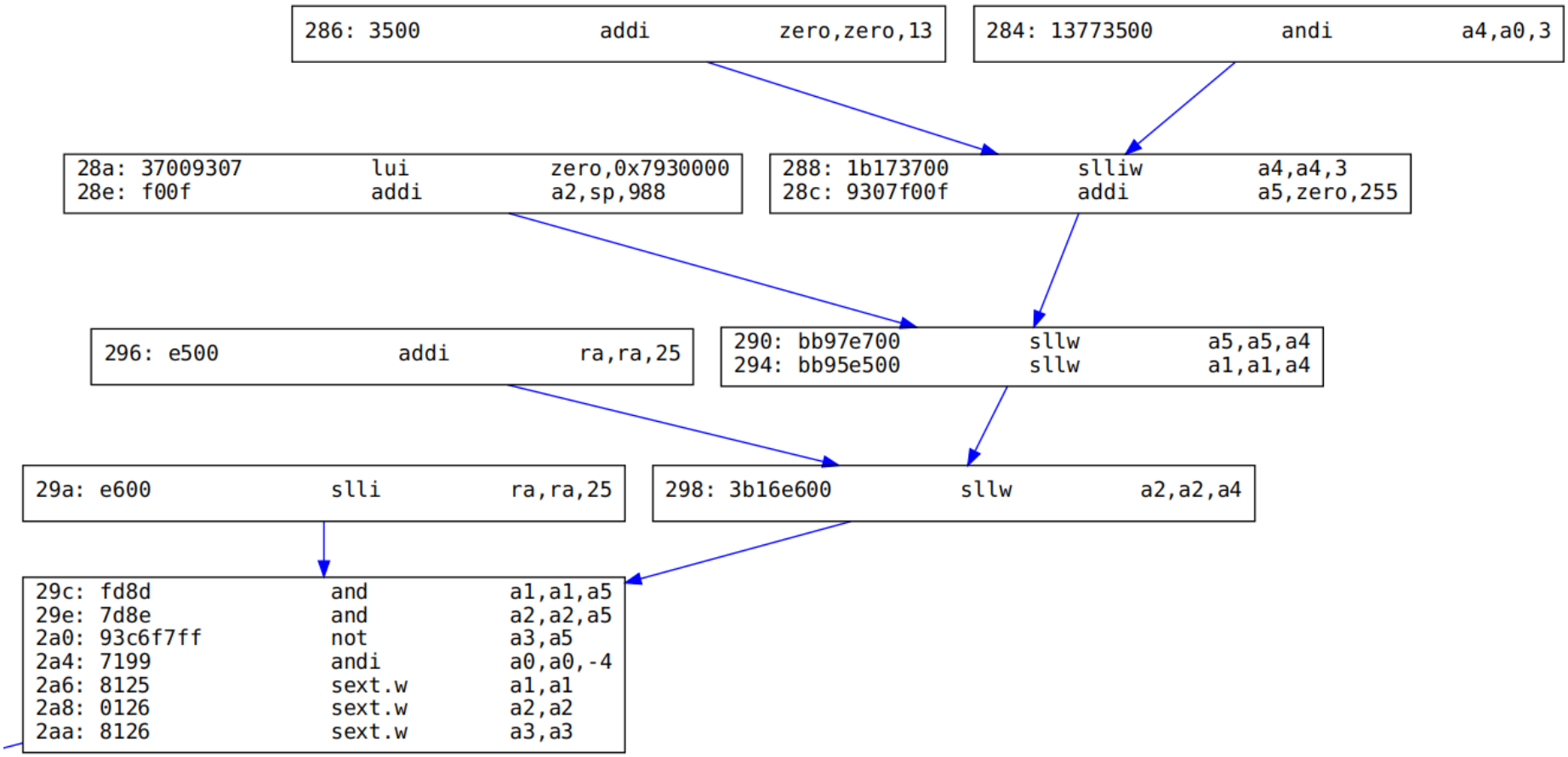
$G \stackrel{\text{def}}{=} (V, E)$;

$End \stackrel{\text{def}}{=} \emptyset$;

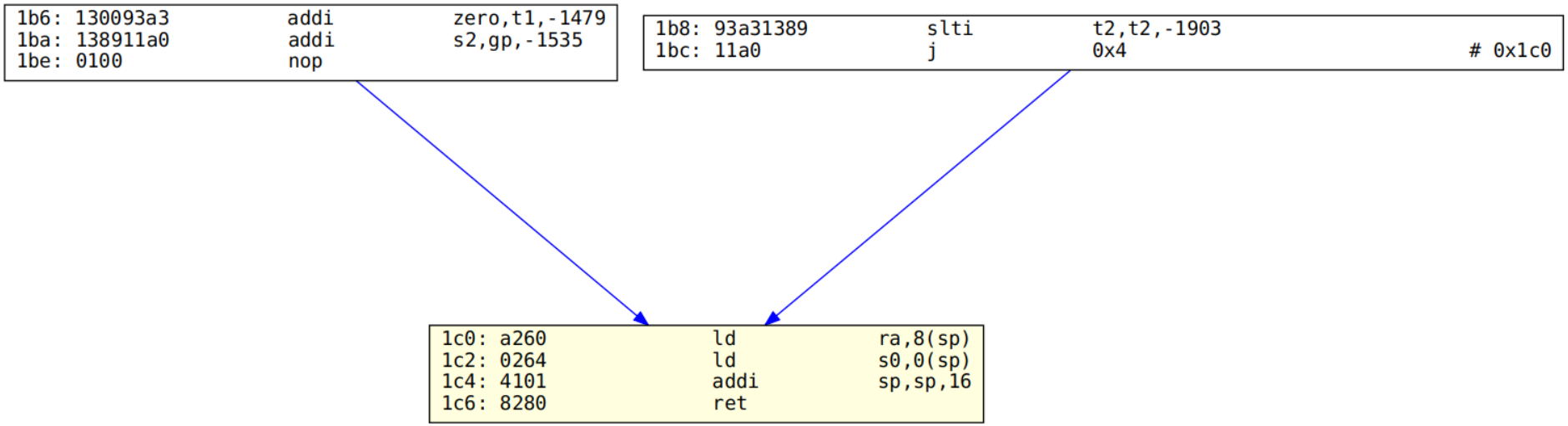
for $pc \stackrel{def}{=} 0$ **to** n **do**
    I := Disasm_one_inst(B_{pc}, ...);
    if I **is not** a valid instruction **then**
        continue
    end
    V.insert(pc);
    foreach pc' **in** I.get_next_pc() **do**
        E.insert(pc, pc')
    end
    if I **is** an indirect jump **then**
        End.insert(pc)
    end
end

$G' \stackrel{\text{def}}{=}$ coreachable(G, End) ;
return G';

Resultat


286: 3500 addi zero,zero,13
284: 13773500 andi a4,a0,3
28a: 37009307 lui zero,0x7930000
28e: f00f addi a2,sp,988
288: 1b173700 slliw a4,a4,3
28c: 9307f00f addi a5,zero,255
296: e500 addi ra,ra,25
290: bb97e700 sllw a5,a5,a4
294: bb95e500 sllw a1,a1,a4
29a: e600 slli ra,ra,25
298: 3b16e600 sllw a2,a2,a4
29c: fd8d and a1,a1,a5
29e: 7d8e and a2,a2,a5
2a0: 93c6f7ff not a3,a5
2a4: 7199 andi a0,a0,-4
2a6: 8125 sext.w a1,a1
2a8: 0126 sext.w a2,a2
2aa: 8126 sext.w a3,a3

Resultat


1b6: 130093a3 addi zero,t1,-1479
1ba: 138911a0 addi s2,gp,-1535
1be: 0100 nop
1b8: 93a31389 slti t2,t2,-1903
1bc: 11a0 j 0x4 # 0x1c0
1c0: a260 ld ra,8(sp)
1c2: 0264 ld s0,0(sp)
1c4: 4101 addi sp,sp,16
1c6: 8280 ret

Next step : le CFG !

Definition (Graphe de flot de contrôle)

Représentation sous forme de graphe (d'une partie) des chemins d'exécution possibles dans un programme.

Next step : le CFG !

Definition (Graphe de flot de contrôle)

Représentation sous forme de graphe (d'une partie) des chemins d'exécution possibles dans un programme.

Démo : Option graphe dans Ghidra/IDA.

Retour sur nos abstractions

Nous avions vu en compilation la notion de fonction :

- Si table des symboles, on a les adresses et leur nom $\rightarrow$ bingo.
- Si l'ABI est connue, "relativement" facile à les retrouver.
- On peut se baser sur les sauts indirects inconditionnels (returns), ou les instructions call (ou similaire en RISC) $\rightarrow$ signatures de fonction.
- On peut sinon regarder la tête du CFG.

Readelf/strip : TP

- Utiliser `readelf` sur le fichier de test
- Utiliser `strip` sur le fichier de test
- Réessayer `readelf`

Détection de fonctions

Compiler-Agnostic Function Detection in Binaries, Andriesse et al.

Constat :

- 25% de faux positifs dans IDA Pro avec les méthodes par signature (certains utilisent du ML pour apprendre les signatures).
- Ce qui définit une fonction : point d'entrée, corps.
- Difficultés : fonctions non contiguës, fonctions à multiples point d'entrée, padding code, données au milieu du code, code mort, tail call, prologues et épilogues alternatifs.

Détection de fonctions

Compiler-Agnostic Function Detection in Binaries, Andriesse et al.

Constat :

- 25% de faux positifs dans IDA Pro avec les méthodes par signature (certains utilisent du ML pour apprendre les signatures).
- Ce qui définit une fonction : point d'entrée, corps.
- Difficultés : fonctions non contiguës, fonctions à multiples point d'entrée, padding code, données au milieu du code, code mort, tail call, prologues et épilogues alternatifs.

```
foo:
    call B
    call A
    ret
```

```
foo:
    call B
    jmp A
```

Algo : Nucleus

1. Créer le CFG
2. Couper au niveau des instructions call ou similaires.
3. Trouver les composantes faiblement connexes
4. Trouver les entry points
   - Directement (instruction call)
   - En récupérant le nœud source d'une composante connexe
   - En trouvant le bloc qui n'a que des sauts arrière
   - En prenant la plus petite adresse

Algo : Nucleus

1. Créer le CFG
2. Couper au niveau des instructions call ou similaires.
3. Trouver les composantes faiblement connexes
4. Trouver les entry points
   - Directement (instruction call)
   - En récupérant le nœud source d'une composante connexe
   - En trouvant le bloc qui n'a que des sauts arrière
   - En prenant la plus petite adresse

- Faux positifs : imprécision sur l'offset des jumps indirects (eg pour les switch case)
- Faux négatifs : tailcall

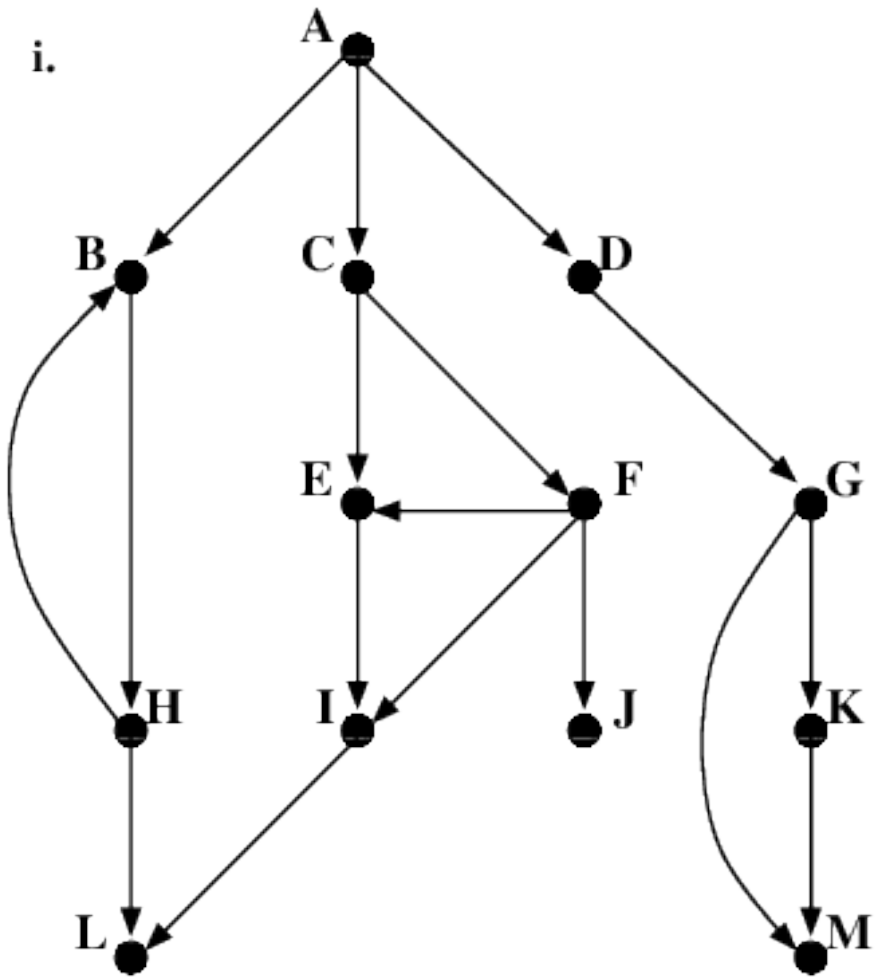
Continuons : les boucles

- Besoin de redéfinir le concept de boucle en termes de graphes.
- Tous les cycles dans un graphe ne constituent pas des boucles.
- Intuitivement : un seul point d'entrée, et un cycle créé par les arcs.

Definition (Dominateur)

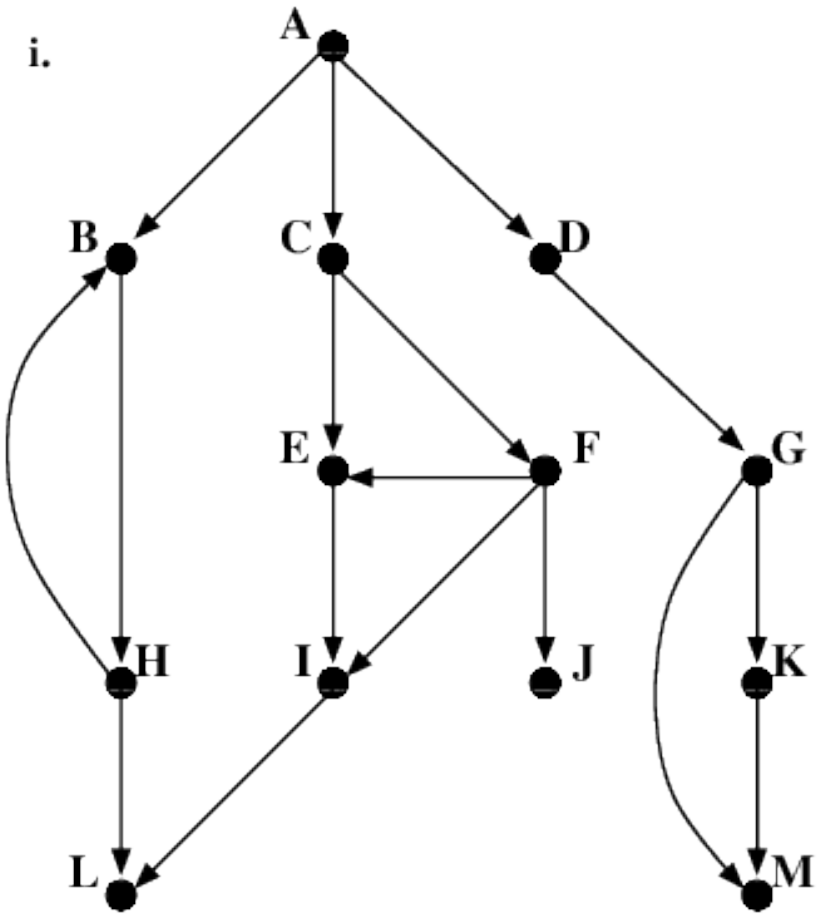
Un nœud d **domine** un nœud n quand tous les chemins partant de la racine à n passent par d.
On dit de plus que d **domine immédiatement** n si aucun nœud q sur un chemin de d à n domine n.

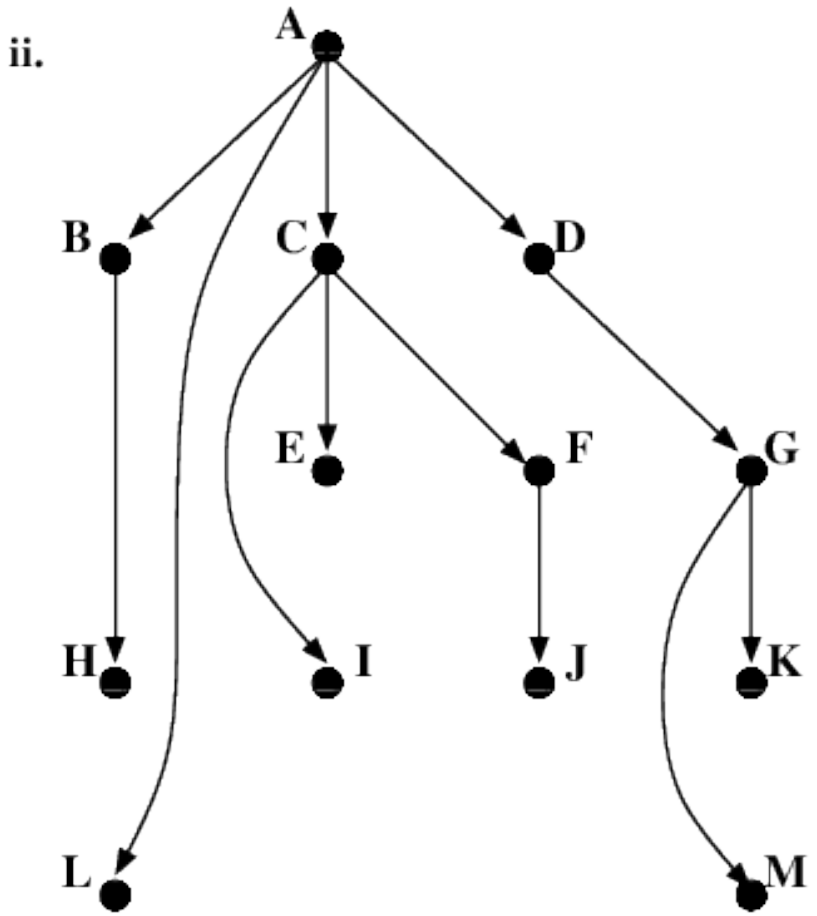
Exemple/TD



Calculer la relation de dominance immédiate de ce graphe (se présente sous forme d'un graphe de dominance).

Exemple/TD : Solution





Défition d'une boucle

- Entrée de la boucle : nœud qui domine tous les nœuds de la boucle
- Arc retour : arc dans lequel la destination domine la source.

Chaque arc retour définit au moins une boucle.

Definition (Boucle naturelle d'un arc retour)

La boucle naturelle d'un arc retour est le plus petit ensemble de nœuds comprenant la source et la destination de l'arc, ainsi que tous les nœuds des chemins allant de la destination à la source.

Algorithme des boucles naturelles

- Construire la relation de dominance dans le graphe
- Trouver les arcs retour
- Calculer les boucles naturelles de chaque arc retour

Exo TD

- Retrouver les boucles naturelles du programme dans Ghidra.
- Vérifier que le décompilateur respecte ces boucles naturelles.

Continuons les abstractions : def-use chain

Definition (def-use chain)

Structure de donnée qui explicite, pour chaque affectation de variable, l'ensemble des locations de programmes où cette variable est utilisée.

Attention

Def-use chain != use-def chain.

Exo TD

- Avec ghidra, tester les def-use chains.
- Tester ensuite le forward et backward slice.
- A quoi servent chacune de ces fonctionnalités ?

Abstrayons un peu plus : les représentations intermédiaires

- Déjà vu en compilation, on refait la même chose mais dans l'autre sens.
- Souvent rencontrés en reverse :
  - SSA (Cytron 91) et variantes (Lapkowski 98, Vanbroekhoven 02, ...)
  - IDA (Guilfanov 2008)
  - Bitblaze (Song et al. 2008)
  - REIL (Dullien et Porst, 2009)
  - BINCOA (Bardin et al. 2011)
  - Miasm (Desclaux 2012)
  - Angr (Shoshitaishvili et al. 2016)
- Gros avantage : indépendant de l'architecture
- Gros avantage : permet de lancer des analyses plus poussées sur un modèle plus propre du code (effets de bords explicités, ...).

Encore une autre abstraction : le typage

Definition (Type)

Un type est une information associée à un élément de programme (donnée, fonction, variable), permettant de renseigner sur la nature de cet élément.

Remarque

On en reparlera dans le cours sur les méthodes formelles.

Quels typages ?

Type Inference on Executables, J. Caballero and Z. Lin

Un peu de contexte :

- On ne dispose que du binaire.
- On souhaite retrouver comment les données sont stockées, manipulées, et interprétées.
- De nombreuses analyses sont en fait du typage : ABI, taille des pointeurs, endianness, UTF-16... Mais des algos spécifiques permettent de les retrouver.
- Même le désassemblage peut être vu comme un problème de typage (distinguer code/donnée) !

Pourquoi typer ? 1/2

- Rétro-ingénierie : trouver un ersatz aux symboles de debug (types, portée des variables, méthodes/sous-méthodes, prototypes de fonctions)
- Décompilation : retrouver les informations de type originales (dépend du langage source)
- Réécriture de code binaire : identifier les pointeurs relatifs ou absolus
- Réutilisation de code binaire : trouver les prototypes pour les interfacer avec le code maison
- Rétro-ingénierie de protocoles : trouver le format des messages demande souvent de typer un buffer.

Pourquoi typer ? 2/2

- Recherche de vuln : trouver la taille des buffers, trouver les pointeurs contenant des adresses de retour, trouver des pointeurs de fonctions.
- Hooking : retrouver le prototype des fonctions
- Analyse forensique : trouver les symboles de debug et les structure de données classiques d'un kernel pour identifier un rootkit.
- Manipulation de programme : cheatengine++
- Fingerprinting : lutter contre le polymorphisme en identifier une abstraction commune aux différentes traces/programmes vus.

Différentes méthodes de typage

Quelques statistiques :

- 38 méthodes différentes de 1999 à 2015 (dont 20 statiques)
- Typant soit directement du binaire, soit une représentation intermédiaire
- (on oublie le désassemblage)
- Avec des méthodes statiques (plutôt reverse) ou dynamiques (plutôt malware) ou concoliques.

Value-based type inference

- Analyse les valeurs stockées dans les registres et la mémoire
- Ne requiert pas de regarder le code
- **Beaucoup** d'heuristiques
- Utile pour trouver les pointeurs et les chaînes de caractères
  - Regarder si la valeur du registre correspond à une adresse valide dans la section data ou le tas
  - Si la donnée est faite de caractères affichables/UTF-16
  - ...
- Combinée avec d'autres méthodes pour augmenter la précision (hauteur de pile, graphe mémoire, ...)

Flow-based type inference

- Beaucoup plus commun. Deux principaux outils :
  - Propagation de type : `a=b`
  - Contraintes de type : `imul eax, ebx`
  - Les deux à la fois : `mov eax, [ebx]`
- On a gratuitement un tas d'information :
  `size_t strlen (const char*)`
- Même problème de complétude vs correction : `memcpy`
- Nécessite une analyse d'alias de pointeurs (méthodes formelles...).
- Résolution des contraintes :
  - Online : à chaque point de programme, on dispose des types de chaque variable $\rightarrow$ pratique pour connaître une erreur de typage.
  - Offline : on collecte toutes les contraintes, et on balance tout dans un solveur. Rate très souvent (casts, unions, ...)

Quelques pièges

- Unification `mov eax ebx` :
  - Quizz : on unifie `eax` et `ebx` ?

Quelques pièges

- Unification `mov eax ebx` :
  - Quizz : on unifie `eax` et `ebx` ?
  - Réponse : ça dépend...

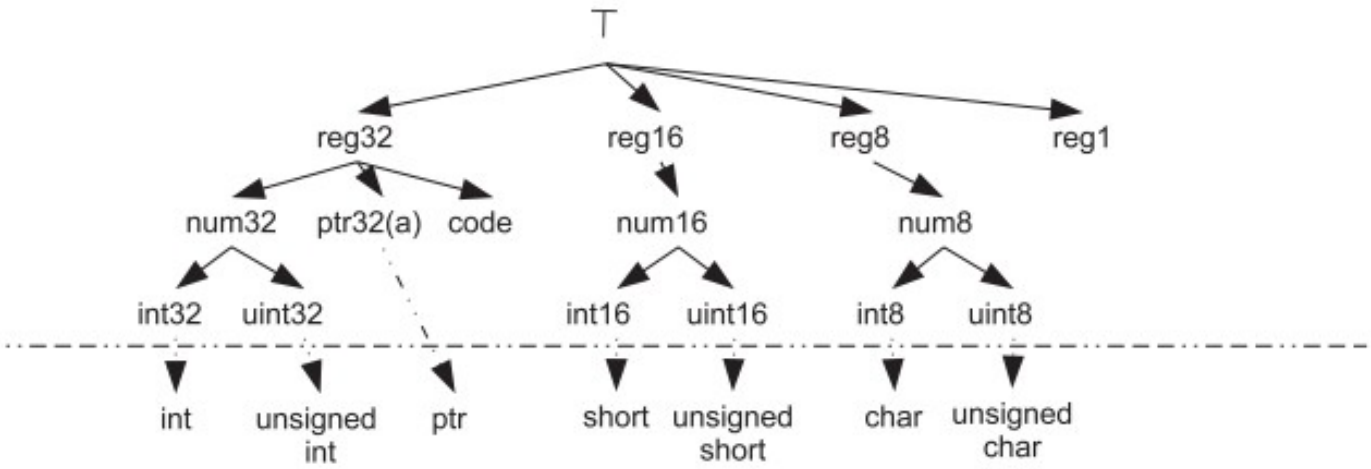
Quelques pièges

- Unification `mov eax ebx` :
  - Quizz : on unifie `eax` et `ebx` ?
  - Réponse : ça dépend...
  - Unifier uniquement les sources (implicit SSA)
  - Unifier lorsque la destination n'est pas un registre.

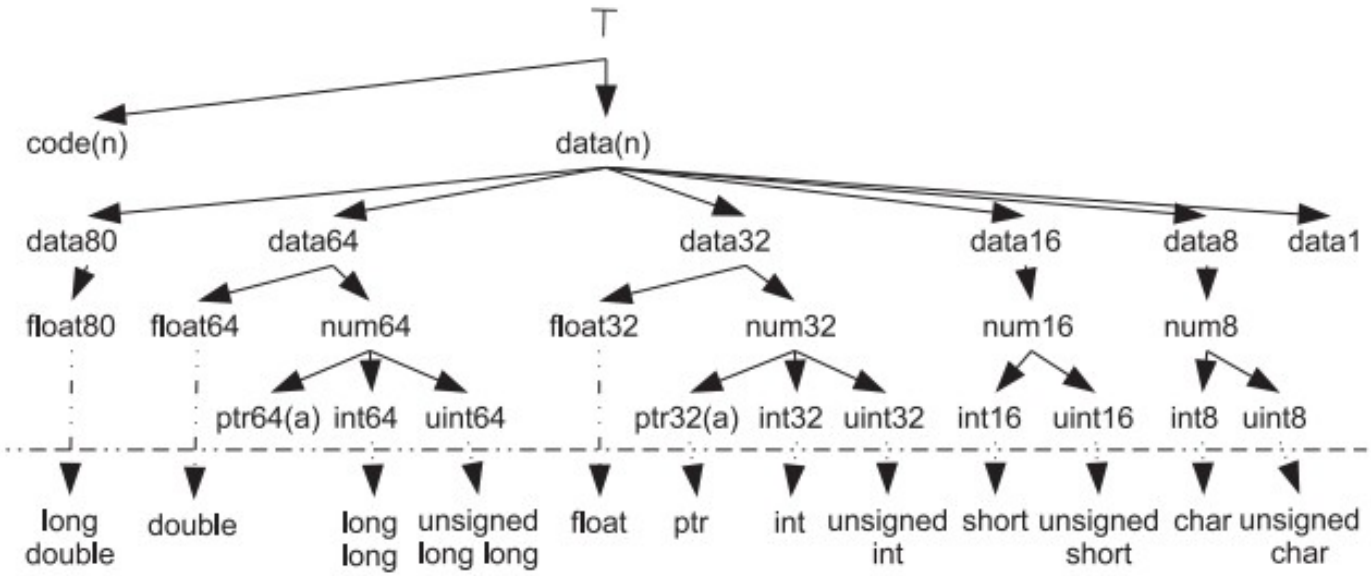
Quelques pièges

- Unification `mov eax ebx` :
  - Quizz : on unifie `eax` et `ebx` ?
  - Réponse : ça dépend...
  - Unifier uniquement les sources (implicit SSA)
  - Unifier lorsque la destination n'est pas un registre.
- Soucis avec le RISC : les instructions ont trop peu de sémantique (par exemple charger une constante avec 4 instructions).
- Fonctions : possibilité d'utiliser le nom des arguments comme type personnalisé.
- Pattern d'accès mémoire : permet de révéler les structures/tableaux...

Exemple de types primitifs retrouvés



a) TIE primitive lattice



b) ARTISTE primitive lattice

Types plus complexes : tableaux et structures

- Trouver des accès mémoires à partir d'un pointeur constant + offsets
- Si l'offset dépend d'une boucle, alors tableau
- Trouver la taille à coup d'heuristiques
- Trouver les types récursifs avec de la *shape analysis* (méthodes dynamiques uniquement).
- Mais quelques soucis avec les structures globales ou en pile (offset depuis esp ou pas d'offset du tout), et autres soucis avec tableaux multidimensionnels, SIMD, padding, strings, ...

Types encore plus complexes : classes

Ce qu'il y a à retrouver :

- Les méthodes
- La hiérarchie des classes (héritage, héritage multiple)
- Méthodes virtuelles et tables virtuelles (en scannant la mémoire pour trouver des tableaux de pointeurs de fonction)
- Représentation mémoire des champs d'une classe (comme pour les structures).
- Mécanismes d'exception.

Pas de solution miracle : utiliser **beaucoup** l'ABI (surtout pour retrouver `this`). Parfois quelques infos supplémentaires grâce à des analyses dynamiques.

Ce que personne fait (pour l'instant)

- Méthode statique de shape analysis
- Types synonymes (même si la méthode des noms de paramètres semble s'en approcher).
- Types abstraits de données (snif').
- Fonctions variadiques/Détection des paramètres inutilisés
- Exploiter les format string.
- Typer des programmes obfusqués.

Fin

Des questions ?

Pour approfondir/bibliographie :
Regarder les papiers cités dans les transparents.