angr

Introduction au cadriciel d'analyse de programmes binaires

Philippe Pépos Petitclerc

UQAM - INF889A

Graphes de flot de contrôle

Autres analyses

Exploration

BitVectorSymbols (BVS), Claripy et SMT Solver

Hooks et simprocs

Veritesting

Vulnerability Discovery

Automatic Exploit Generation

Graphes de flot de contrôle

CFGFast

- Recouvrement récursif (*linear descent*)
- Résolution des branchements indirects

CFGEmulated

- Émulation de chaque fonction à partir du début
  - Niveau de contexte variable (Appelé, appelant)
- *Backwards slicing*, *Symbolic back traversal*

Autres analyses

Voir le source...

github.com/angr/angr

angr-utils

Librairie qui permet de tracer (*plot*) les graphes résultants des analyses.

Exploration

Exploration de `fauxware`

Démonstration

BitVectorSymbols (BVS), Claripy et SMT Solver

BitVectorSymbols (BVS)

- Représenter des vecteurs de bits
- ... et les opérations dessus

```
>>> one = state.solver.BVV(1, 64)
>>> one
<BV64 0x1>
>>> one_hundred = state.solver.BVV(100, 64)
>>> one_hundred
<BV64 0x64>
>>> one + one_hundred
<BV64 0x65>
>>> one_hundred - one*200
<BV64 0xffffffffffffff9c>
```

SMT Solver

- Par défaut le solveur utilisé est *Z3* (*Microsoft Research*)
- Permet de résoudre des systèmes
  - Donner un ensemble de valeurs satisfaisant un système de contraintes
  - Exemple: démonstration

Claripy

- Claripy est l'engin (abstraction) de résolution de systèmes d'`angr`.
- L'objectif est de présenter un API plus proche des concepts d'`angr`.
- Présente plusieurs *backends*:
  - Solver (SMT)
  - SolverVSA (Value-Set Analysis)
  - SolverReplacement (*Custom*)
  - etc.

Contraintes manuelles
On peut ajouter des contraintes additionnelles manuellement.

Démonstration.

Hooks et simprocs

Procédures fournises

On peut remplacer l'exécution de certaines fonctions par des implémentations en python. `angr` en fourni quelques unes.

```
p.hook(0x422690, angr.SIM_PROCEDURES['libc']['memcpy']())
p.hook(0x408F10, angr.SIM_PROCEDURES['libc']['puts']())
```

Un API existe également pour implémenter des appels système.
(Voir le code)

Veritesting

Exécution symbolique statique (ESS ou SSE)

Technique de vérification formelle dans laquelle on représente l'exécution d'un programme comme une formule logique.

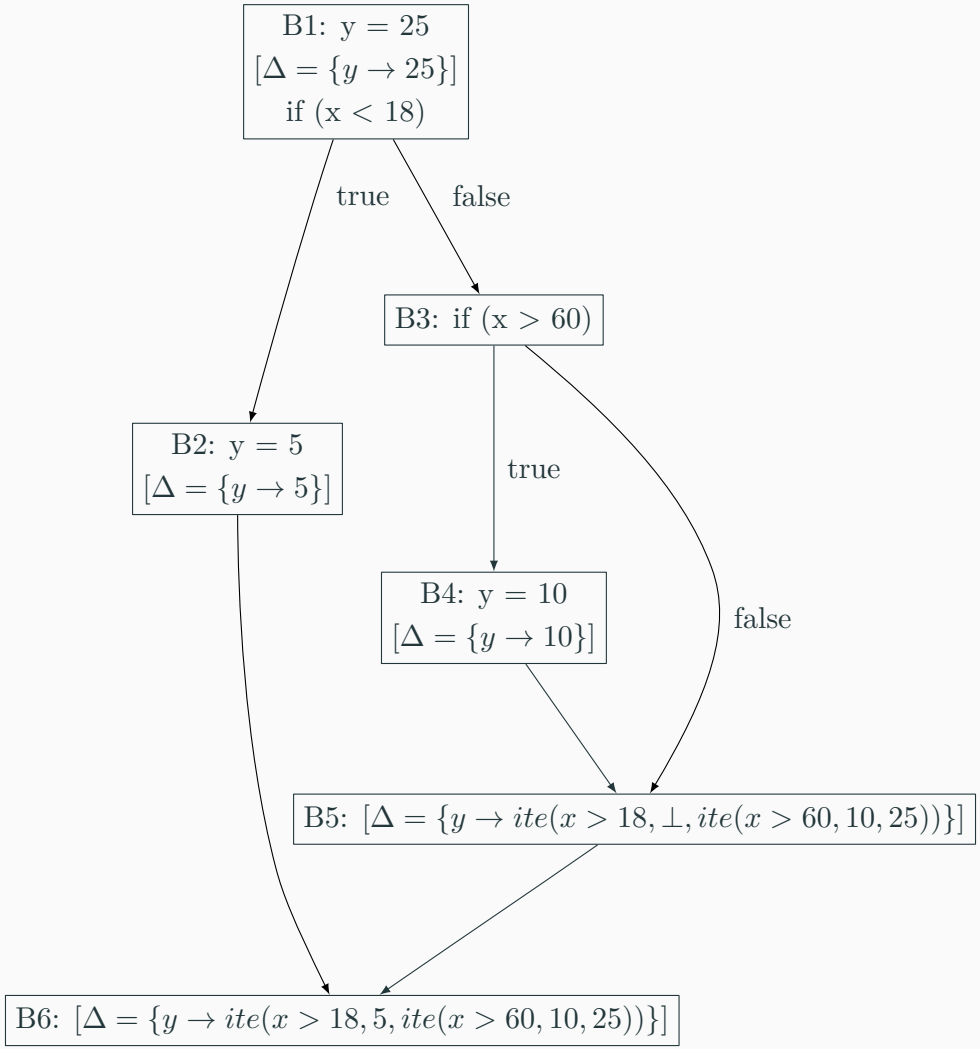
- Même formule logique pour tout le programme (tous les chemins d'exécution du programme)

Exemple de code

```
y = 25;
if (x < 18) {
        y = 5;
} else if (x > 60) {
        y = 10;
}
```

Exécution symbolique statique (Suite)

Visualisation de l'ESS



ESS: Avantages et inconvénients

Avantages:

- Synthèse des différents chemins d'exécution au point de confluence
  - Gains de performance (formule compacte)

Inconvénients:

- Solveur doit résoudre des formules complexes

Exécution symbolique dynamique (ESD ou DSE)

- S'effectue lors de l'interprétation du programme
- Une branche d'exécution à la fois
- Génère des prédicats de chemins

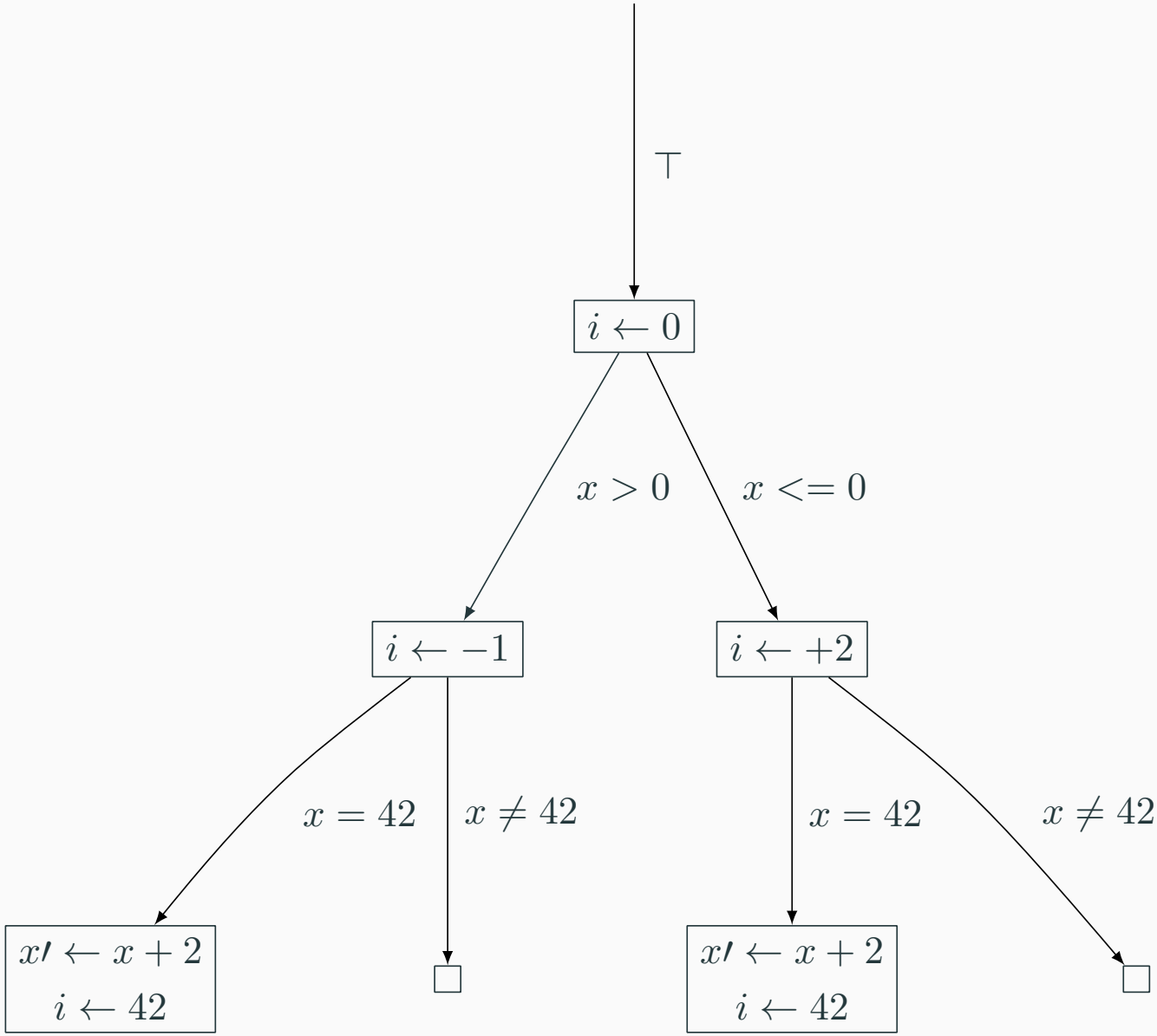
ESD: Construction du prédicat de chemin

- Exécute jusqu'au branchement
- *Fork* au branchement
- Ajoute au prédicat de chemin la condition pour suivre la branche *vrai*
    - L'exécuteur *forké* ajoutera le complément à son pdc

ESD: Exemple

```
int i = 0;
int x = read_int();
if (x > 0) {
    i = i - 1;
else {
    i = i + 2;
}
if (x == 42) {
    x = x + i
    i = 42;
}
```

ESD: Exemple (suite)



Avantages

- Simple à implémenter
- Résolution rapide des prédicats de chemins (formules plus simples)
- Grande reproductibilité des résultats
- Contournement facile du problème de la complétion (Analyse locale)
  - Exécuter concrètement
  - Simuler
  - Substitué
  - Ignoré

Limitations

Explosion combinatoire de chemins (et donc d'exécuteurs)

Limitations

Explosion combinatoire de chemins (et donc d'exécuteurs)

```
int counter = 0 , values = 0;
for (i = 0; i < 100; i ++) {
        if (input[i] == 'B') {
                counter++;
                values += 2;
        }
}

if (counter == 75)
        bug();
```

Limitations

Explosion combinatoire de chemins (et donc d'exécuteurs)

```
int counter = 0 , values = 0;
for (i = 0; i < 100; i ++) {
        if (input[i] == 'B') {
                counter++;
                values += 2;
        }
}

if (counter == 75)
        bug();
```

2^{100} chemins d'exécution possible. 2 sont utiles.

Veritesting (très résumé)

Profiter de la synthèse des branchements de l'exécution symbolique statique afin de déterminer quels exécuteurs sont réellement nécessaires pour couvrir tous les chemins d'exécution.

Veritesting (très résumé)

Profiter de la synthèse des branchements de l'exécution symbolique statique afin de déterminer quels exécuteurs sont réellement nécessaires pour couvrir tous les chemins d'exécution.

- DSE jusqu'à un branchement
- Passe de transformation et d'augmentation du GFC
- Déterminer la frontière difficile à modéliser
- SSE
- Créer les exécuteurs avec les nouveaux pdc
- DSE

Vulnerability Discovery

Automatic Exploit Generation