INF889A

Analyse de programmes pour la sécurité logicielle

Chapitre 3 - Exécution symbolique

Quentin Stiévenart

Hiver 2024

Introduction

Couverture d'un programme

On aimerait bien couvrir le programme dans son entiereté

Quelle mesure de couverture devrait-on utiliser ?

- Couverture de *lignes*
- Couverture de *branches*
- Couverture *dataflow*
- Couverture de chemins

Couverture de lignes

code/abs.c

```c
int abs(int x) {
    if (x < 0) {
        return -x;
    }
}

int main(int argc, char **argv) {
    assert(abs(-5) == 5);
    return 0;
}
```

Couverture de lignes

code/abs.c

```
int abs(int x) {
    if (x < 0) {
        return -x;
    }
}
```

Problème : `abs(5)` est non défini

Couverture de lignes

code/abs.c

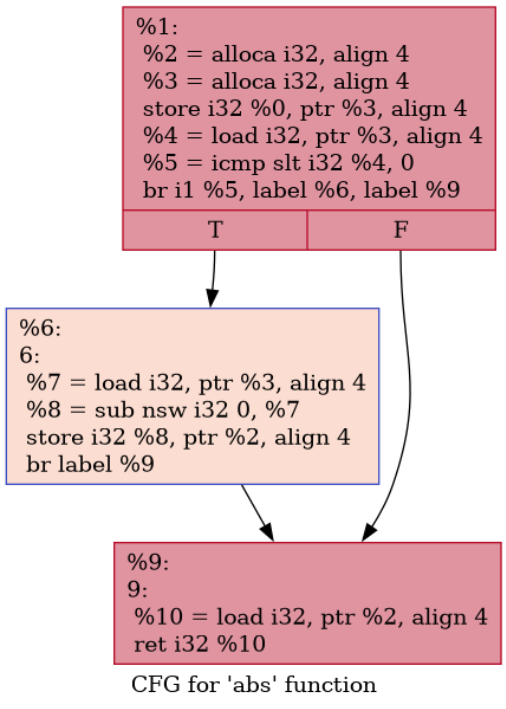
```
$ clang abs.c -o abs -fprofile-instr-generate -fcoverage-mapping
$ ./abs
$ llvm-profdata merge -sparse default.profraw -o default.profdat
$ llvm-cov show ./abs -instr-profile=default.profdata
```

Toutes les lignes sont couvertes.

CFG

code/abs.c

```
$ clang -S -emit-llvm abs.c -o abs.ll \
    -Xclang -disable-O0-optnone -fno-discard-value-names
$ opt abs.ll -disable-output -passes=dot-cfg
```



CFG for 'abs' function

On a couvert toutes les instructions, mais pas tous les arcs

$\rightarrow$ on va préférer la couverture de branches.

Couverture de branches

code/contains.c

```c
int contains(int a[], int len, int x) {
    int found = 0;
    for (int i = 0; i <= len; i++) {
        if (a[i] == x) {
            found = 1;
            break;
        }
    }
    return found;
}

int main() {
    int a[] = {1, 2};
    assert(contains(a, 2, 2));
}
```

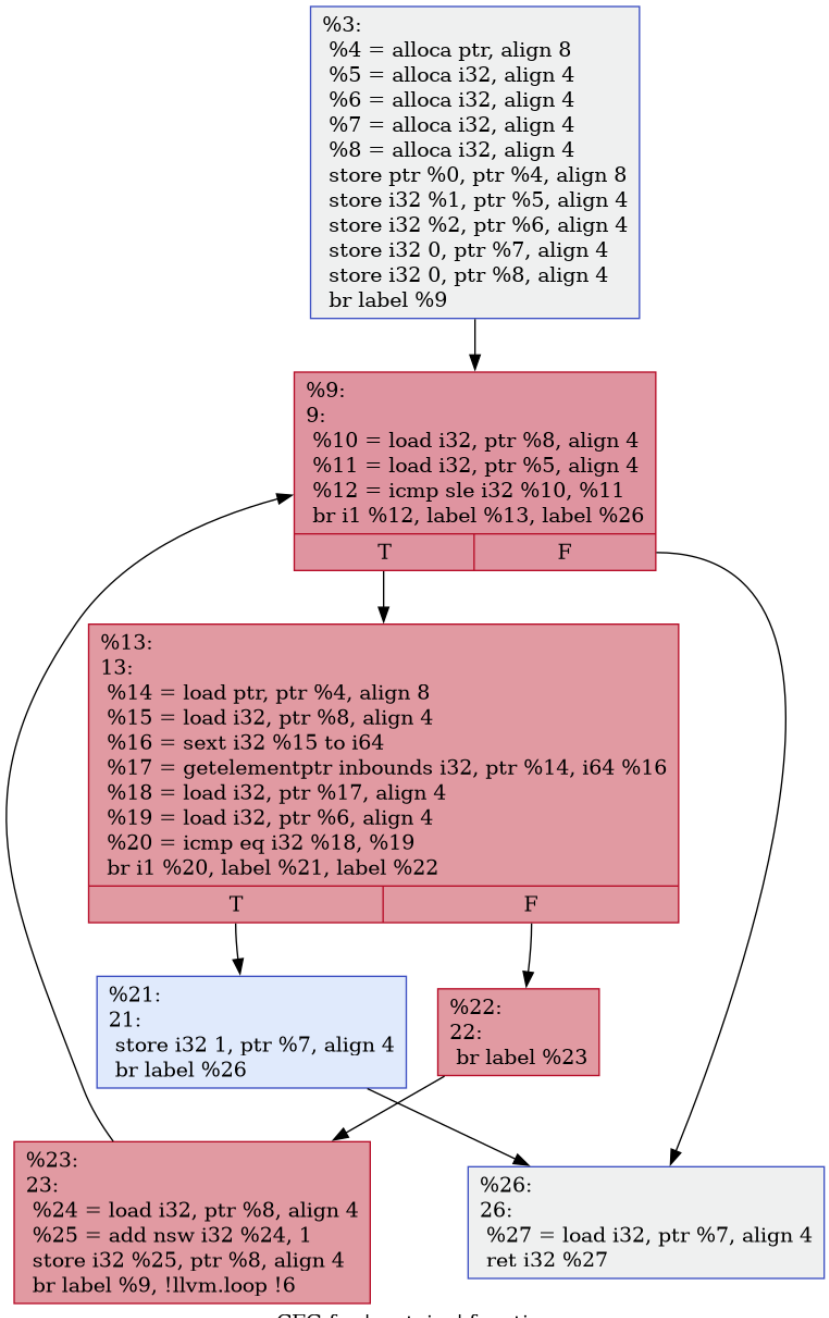
Couverture de branches

code/contains.c

```
$ clang contains.c -o contains \
    -fprofile-instr-generate -fcoverage-mapping
$ ./contains
$ llvm-profdata merge -sparse default.profraw -o default.profdat
$ llvm-cov report ./contains -instr-profile default.profdata

$ clang -S -emit-llvm contains.c -o contains.ll \
    -Xclang -disable-O0-optnone -fno-discard-value-names
$ opt contains.ll -disable-output -passes=dot-cfg
```

Couverture de branches

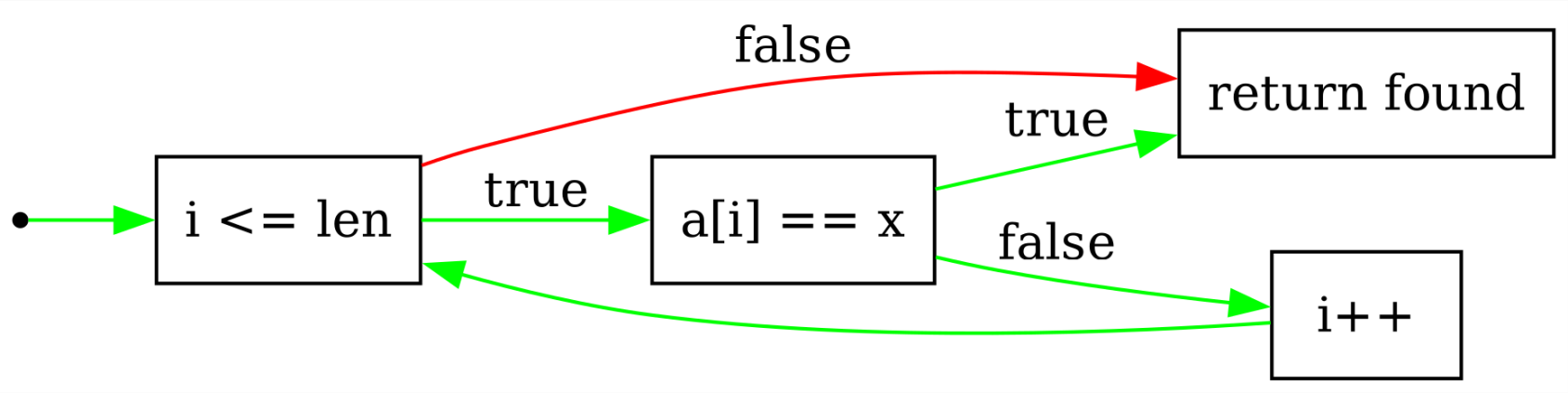
code/contains.c



Couverture de branches

code/contains.c

Couverture avec {1,2} et 2 : 3 / 4 branches couvertes



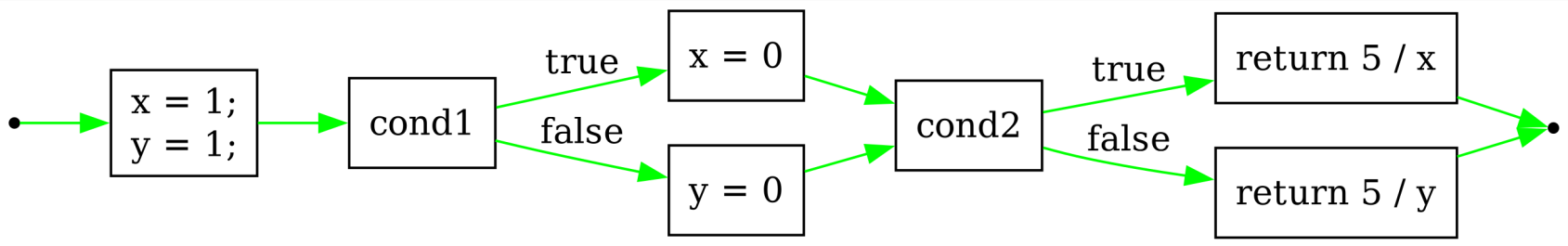
Couverture de branches

code/branchy.c

```c
int foo(int cond1, int cond2) {
    int x = 1;
    int y = 1;

    if (cond1) x = 0;
    else y = 0;

    if (cond2) return 5 / x;
    else return 5 / y;
}

int main() {
    assert(foo(1, 0) >= 0);
    assert(foo(0, 1) >= 0);
}
```

Couverture de branches

code/branchy.c



100% de couverture de branche, mais pas suffisant pour trouver l'erreur

Couverture de chemin

`code/branchy.c`

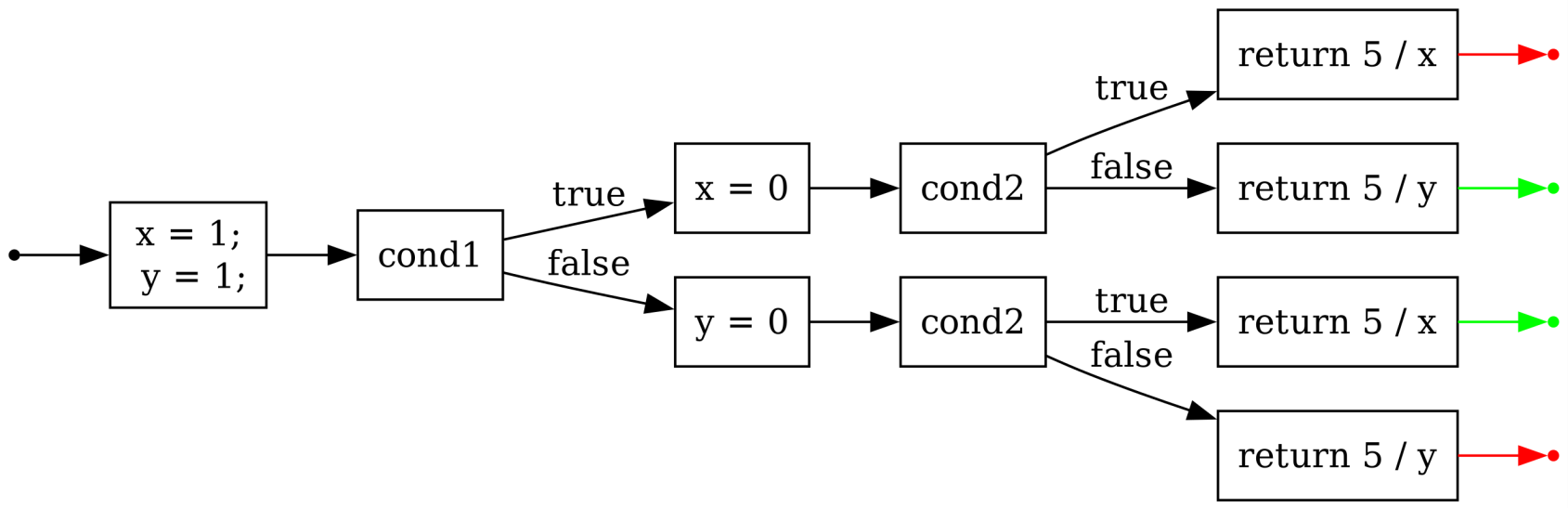
On a 4 chemins en tout :

- `a = false, b = false`
- `a = false, b = true`
- `a = true, b = false`
- `a = true, b = true`

Arbre d'exécution

code/branchy.c

On peut représenter cela par un arbre :



Exécutions infinies

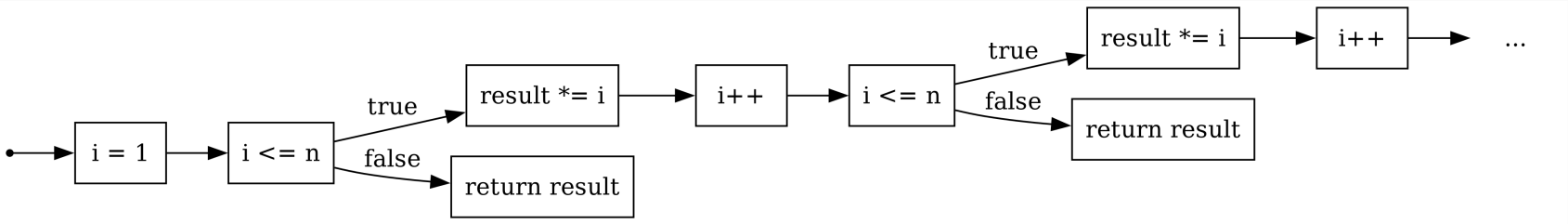
code/fact.c

```c
#include <stdlib.h>

int main(int argc, char *argv[]) {
    int n = atoi(argv[1]);

    int result = 1;
    for (int i = 1; i <= n; i++) {
        result *= i;
    }
    return result;
}
```

Exécutions infinies

code/fact.c

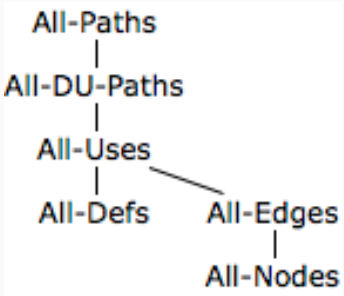
Nombre de chemins non déterminé statiquement, ou même infini



Couverture

Quel type de couverture souhaite-t-on ?

- En génie logiciel, le *dataflow coverage* donne un bon compromis
- En analyse de programme, on aimerait avoir une approche qui *tend* vers une exploration exhaustive des chemins



Alspaugh, T. Software Testing, 2019

Exécution symbolique statique

King, 1976

King. (1976). Symbolic execution and program testing, CACM.

> *Instead of supplying the normal inputs to a program one supplies symbols representing arbitrary values. The execution proceeds as in normal execution except that values may be symbolic formulas over the input symbols*

Deux ingrédients clés :

1. représentation des chemins par une **condition de chemin** (*path condition*)
2. stockage des valeurs symboliques dans une **mémoire symbolique** (*symbolic store*)

Rappel : limitation du fuzzing

```
if (x == 0x12345678) {
  assert(false);
}
```

Difficile à atteindre par fuzzing, pourtant on sait qu'il faut que `x = 0x12345678`

Exécution symbolique : intuition

code/symex.c

```
void testme(int x, int y) {
    int z = 2*y;
    if (z == x) {
        if (x > y + 10) {
            assert(0);
        }
    }
}
```

Comment atteindre l'assert ?

Exécution symbolique : intuition

code/symex.c

```
void testme(int x, int y) {
    int z = 2*y;
    if (z == x) {
        if (x > y + 10) {
            assert(0);
        }
    }
}
```

On veut :

- `z = x` avec `z = 2*y`, et
- `x > y + 10`

Donc : `2*y = x && x > y + 10`

Résolution de formules symboliques

code/symex.z3

C'est une formule logique : $2y = x \wedge x > y + 10$

On peut la résoudre avec des *solvers* SAT/SMT tels que *Z3*

```
(declare-const x Int)
(declare-const y Int)
(assert (= (* 2 y) x))
(assert (> x (+ y 10)))
(check-sat)
(get-model)
```

Solveurs SMT

Satisfiability modulo theories : généralisation du problème **SAT**, problème **NP-complet**

SAT : formules de type $(x_1 \vee x_2 \vee \cdots) \wedge (y_1 \vee y_2 \vee \cdots)$ sur des variables booléennes

Un solveur :

- permet de dire si une formule est satisfaisable (`sat`/`unsat`)
- permet de trouver des assignations aux variables telle que la formule est satisfaisable

SMT = SAT + types avancés (nombres, listes, ...)

Résolution de formules symboliques

code/symex.z3

```
$ z3 symex.z3
sat
(
  (define-fun y () Int
    11)
  (define-fun x () Int
    22)
)
```

Condition de chemin (*path condition*)

 `code/symex.c`

Idée : on va représenter un chemin d'exécution par une formule symbolique

La formule représente les conditions qui doivent être vraies pour explorer ce chemin

$2b = a \wedge a > b + 10$ est la condition nécessaire pour atteindre l'`assert`

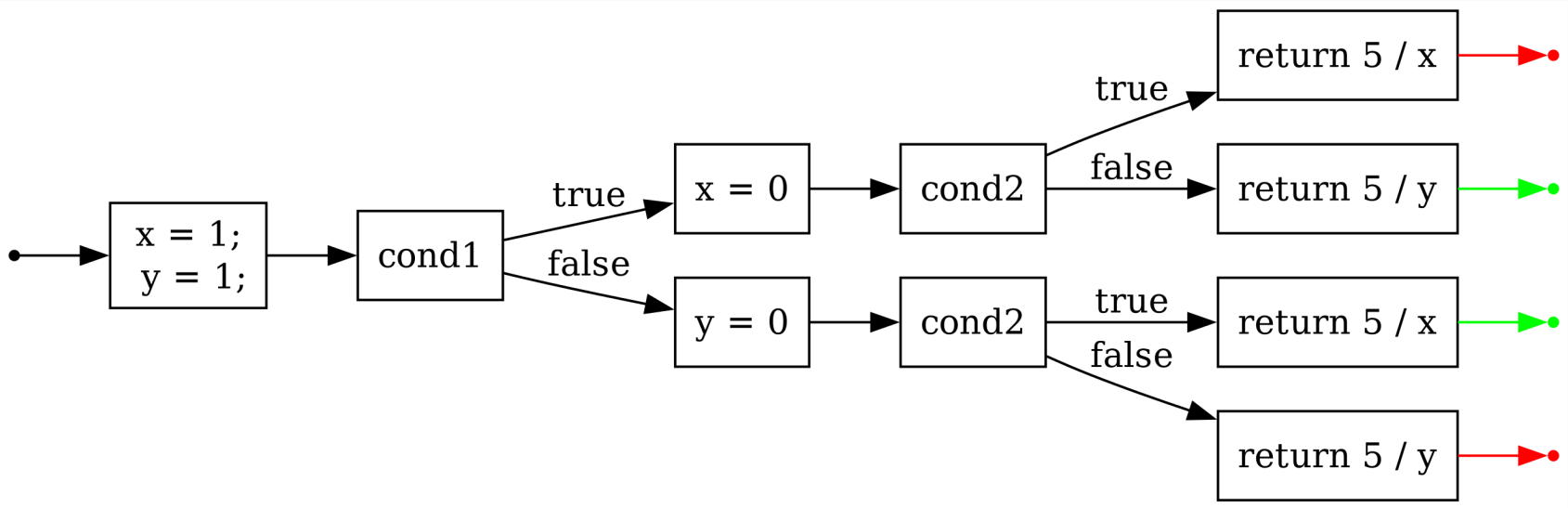
Note : on va représenter les valeurs symboliques par a, b plutôt que par les noms des variables `x` et `y`.

Condition de chemin (*path condition*)

code/branchy.c

```c
int foo(int cond1, int cond2) {
    int x = 1;
    int y = 1;

    if (cond1) x = 0;
    else y = 0;

    if (cond2) return 5 / x;
    else return 5 / y;
}

int main() {
    assert(foo(1, 0) >= 0);
    assert(foo(0, 1) >= 0);
}
```

Condition de chemin (*path condition*) code/branchy.c



4 chemins :

- $a \wedge b$
- $a \wedge \neg b$
- $\neg a \wedge b$
- $\neg a \wedge \neg b$

Condition de chemin (*path condition*)

La condition de chemin au début de l'exécution d'un programme est *true*

$$PC_0 = true$$

À chaque branchement, on y ajoute un terme :

$$PC_1 = true \wedge a$$

$$PC_2 = true \wedge a \wedge b$$

Mémoire d'un programme

On peut représenter mathématiquement la mémoire d'un programme par une fonction :

$$\sigma \in \mathit{Var} \rightharpoonup \mathit{Value}$$

Avec les notations associées :

- Accéder à la valeur d'une variable $\mathtt{x}$: $\sigma(\mathtt{x})$
- Changer la valeur d'une variable x pour devenir v : $\sigma[\mathtt{x} \mapsto v]$
- Dénoter une mémoire : $\{\mathtt{x} \mapsto 1, \mathtt{y} \mapsto 2\}$

Évolution de la mémoire d'un programme

Lors de l'exécution, la mémoire peut être mise à jour à chaque instruction :

$\sigma = \{\mathtt{x} \mapsto 1, \mathtt{y} \mapsto 2\}$

```
int z = x + y;
```

$\sigma' = \{\mathtt{x} \mapsto 1, \mathtt{y} \mapsto 2, \mathtt{z} \mapsto 3\}$

Mémoire symbolique

On ne va pas manipuler des valeurs *concrètes* mais des symboles.
On utilise donc une mémoire *symbolique* :

$$\hat{\sigma} \in \mathit{Var} \rightharpoonup \mathit{Sym}$$

Par exemple :

$$\{\mathtt{x} \mapsto a, \mathtt{y} \mapsto b\}$$

- $\mathtt{x}$ et $\mathtt{y}$ sont des variables
- a et b sont des symboles

Évolution de la mémoire symbolique

On effectue des opérations *symboliques* car on n'a pas accès aux valeurs concrètes

$\hat{\sigma} = \{\mathtt{x} \mapsto a, \mathtt{y} \mapsto b\}$

```
int z = x + y;
```

$\hat{\sigma}' = \{\mathtt{x} \mapsto a, \mathtt{y} \mapsto b, \mathtt{z} \mapsto a + b\}$

On a une variable symbolique par entrée de notre programme. Le reste est propagé sous forme de contraintes.

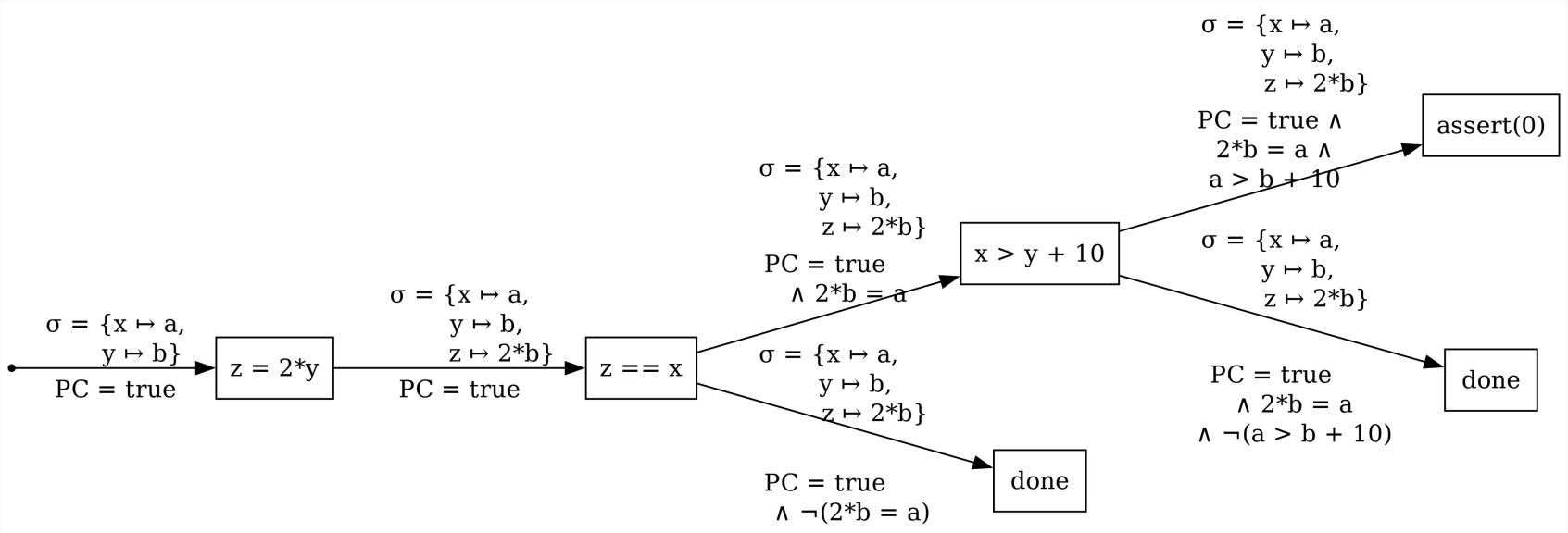
Exécution symbolique statique

code/symex.c

```
void testme(int x, int y) {
    // σ̂ = {x ↦ a, y ↦ b}, PC = true
    int z = 2*y;
    // σ̂ = {x ↦ a, y ↦ b, z ↦ 2 * b}
    if (z == x) {
        // PC = true ∧ 2 * b = a
        if (x > y + 10) {
            // PC = true ∧ 2 * b = a ∧ a > b + 10
            assert(0);
        } else {
            // PC = true ∧ 2 * b = a ∧ ¬(a > b + 10)
        }
    } else {
        // PC = true ∧ ¬(2 * b = a)
    }
}
```

Exécution symbolique statique

code/symex.c



On a donc trois chemins :

$$2 * b = a \wedge a > b + 10$$

$$2 * b = a \wedge \neg(a > b + 10)$$

$$\neg(2 * b = a)$$

Exercice

Soit le programme suivant :

```
x = input();
x = x + 5;

if (x > 0)
    y = input();
else
    y = 10;

if (x > 2)
    if (y == 2789)
        assert(false);
```

Exercice

1. Représenter le CFG de ce programme
2. Combien de chemins ce programme possède-t-il ?
3. Représenter l'arbre d'exécution du programme et annoter ses arcs avec :

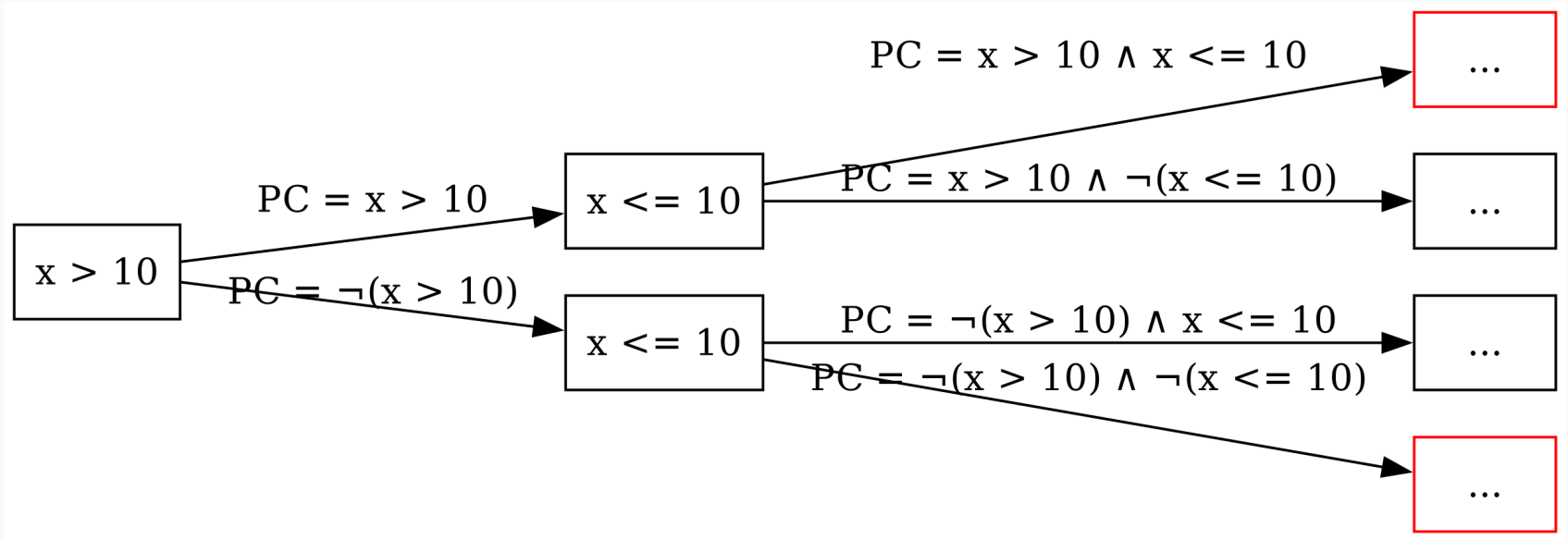
- l'état symbolique
- la condition de chemin

4. Quelle formule logique permet d'atteindre l'`assert` ?
5. Résoudre cette formule

Chemins inatteignables (*infeasible paths*)

```
if (x > 10) { ... }
if (x <= 10) { ... }
```

Chemins inatteignables (*infeasible paths*)



Chemins inatteignables

`code/infeasible.z3`

Ce programme possède quatre chemins, mais deux sont impossibles :

$$x > 10 \wedge x <= 10$$

$$\neg(x > 10) \wedge \neg(x <= 10)$$

Si on essaye de les résoudre avec z3 :

```
$ z3 infeasible.z3
unsat
```

Les formules non-satisfaisables correspondent à des chemins inatteignables

Chemins infinis

Certains programmes peuvent avoir un nombre infini de chemins d'exécution :

```
int x = read_int();
while (x > 0) {
  x = read_int();
}
```

Ou plus simplement :

```
while (true) { ... }
```

En pratique, une exécution symbolique statique va limiter la profondeur de recherche

Explosion combinatoire

Soit la famille de programmes suivante :

```
if (x1) { ... } else { ... }
if (x2) { ... } else { ... }
...
if (xN) { ... } else { ... }
```

Combien de chemins ?

Explosion combinatoire

Soit la famille de programmes suivante :

```
if (x1) { ... } else { ... }
if (x2) { ... } else { ... }
...
if (xN) { ... } else { ... }
```

Combien de chemins ? $\rightarrow 2^n$

En pratique : on explore l'arbre avec un budget (temps, nombre de chemins parcourus) limité et/ou une profondeur limitée

Stratégies d'exploration

Comme on ne sait pas explorer l'arbre en entier, il faut une **stratégie d'exploration**

Plus sur ce sujet plus tard dans le chapitre

Utilisation du solveur SMT en pratique

Résoudre un problème SMT est **NP-complet** : la majorité du temps peut être passé dans le solveur.

Il est possible de *cacher* des solutions intermédiaires :

- $2 * b = a \wedge a > b + 10$ donne une solution
- $2 * b = a$
  - formule plus faible
  - peut **toujours** utiliser la même solution
- $2 * b = a \wedge a > b + 10 \wedge a > 0$
  - formule plus forte
  - peut **parfois** utiliser la même solution (sinon: appel au solveur)

Limite : code hors de notre contrôle

```
if (x == f(y)) { ... } else { ... }
```

`f` peut :

- être du code complexe à analyser (par exemple : venant de la `libc`)
- être du code impossible à analyser statiquement (par exemple : une requête réseau)
- mener à des requêtes non supportées par le solveur SMT (par exemple : chaînes de caractères non supportées par certains solveurs)

Supporter du code hors de notre contrôle

```
if (x == f(y)) { ... } else { ... }
```

On peut :

- Analyser `f`
  - Peut être très complexe ou impossible
- Utiliser une version simplifiée de `f`
  - Par exemple, analyser `musl` plutôt que `glibc`
  - On ne représente plus exactement la sémantique réelle
- Exécuter un modèle de `f`
  - Tâche complexe (exemple : modèle de `read`)

Exécution symbolique statique : garanties

Selon l'hypothèse (irréaliste) que :

- on a exploré tous les chemins
- on ne tombe pas sur du code hors de notre contrôle ou non supporté

L'approche est théoriquement :

- *sound* : toutes les entrées menant à des erreurs sont trouvées
- *complete* : toutes les entrées considérées comme menant à des erreurs mènent bien à des erreurs

Mais toute approche pratique est *unsound* : tous les chemins ne peuvent pas être explorés

Exécution symbolique dynamique

Exécution symbolique dynamique

Godefroid, Klarlund, & Sen. (2005). DART: Directed automated random testing. PLDI.

Cadar, & Sen. (2013). Symbolic execution for software testing: three decades later. CACM

But : supporter le code hors de notre contrôle en se basant sur des observation d'exécution réelles

Exécution symbolique dynamique : concepts clés

Concepts clés :

- on fait rouler le programme
- le programme est instrumenté pour pouvoir maintenir l'état symbolique pendant son exécution
- on explore un chemin à la fois
- on utilise les entrées concrètes si on ne sait pas résoudre la condition de chemin

Mémoire d'un programme (rappel)

On peut représenter mathématiquement la mémoire d'un programme par une fonction :

$$\sigma \in \mathit{Var} \rightharpoonup \mathit{Value}$$

Avec les notations associées :

- Accéder à la valeur d'une variable $\mathtt{x}$: $\sigma(\mathtt{x})$
- Changer la valeur d'une variable x pour devenir v : $\sigma[\mathtt{x} \mapsto v]$
- Dénoter une mémoire : $\{\mathtt{x} \mapsto 1, \mathtt{y} \mapsto 2\}$

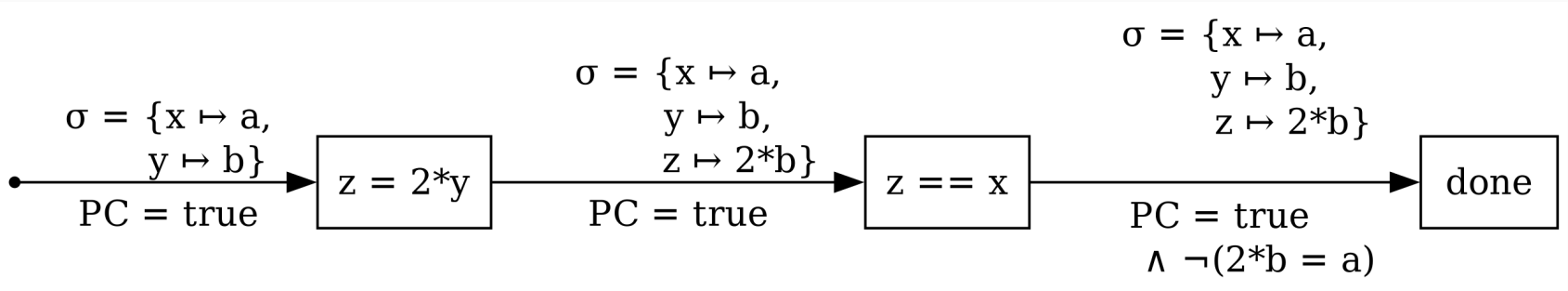
Exécution symbolique dynamique : exemple code/symex.c

Itération 1, avec `x = 22, y = 7` (valeurs aléatoires)

```
void testme(int x, int y) {
    // σ = {x ↦ 22, y ↦ 7}, σ̂ = {x ↦ a, y ↦ b},  PC = true
    int z = 2*y;
    // σ = {x ↦ 22, y ↦ 7, z ↦ 14}, σ̂ = {x ↦ a, y ↦ b, z ↦ 2 * b}
    if (z == x) {
        if (x > y + 10) {
            abort();
        } else {
        }
    } else {
        // PC = true ∧ ¬(2 * b = a)
    }
}
```

Sélection d'entrée pour l'itération suivante

code/symex.c



Avec `x = 22`, `y = 7`, on explore le chemin :

$$true \wedge \neg(2 * b = a)$$

Quelle entrée choisir ensuite ?

Sélection d'entrée pour l'itération suivante

 `code/symex.c`

On prend un chemin exploré et on fait la négation d'une de ses clauses

$$true \wedge \neg(2 * b = a)$$

devient :

$$true \wedge (2 * b = a)$$

Un solveur SMT nous permet de trouver un résultat qui satisfait à cette formule

Sélection d'entrée pour l'itération suivante code/symex.c

```
$ z3 symex-concolic1.z3
sat
(
  (define-fun b () Int
    0)
  (define-fun a () Int
    0)
)
```

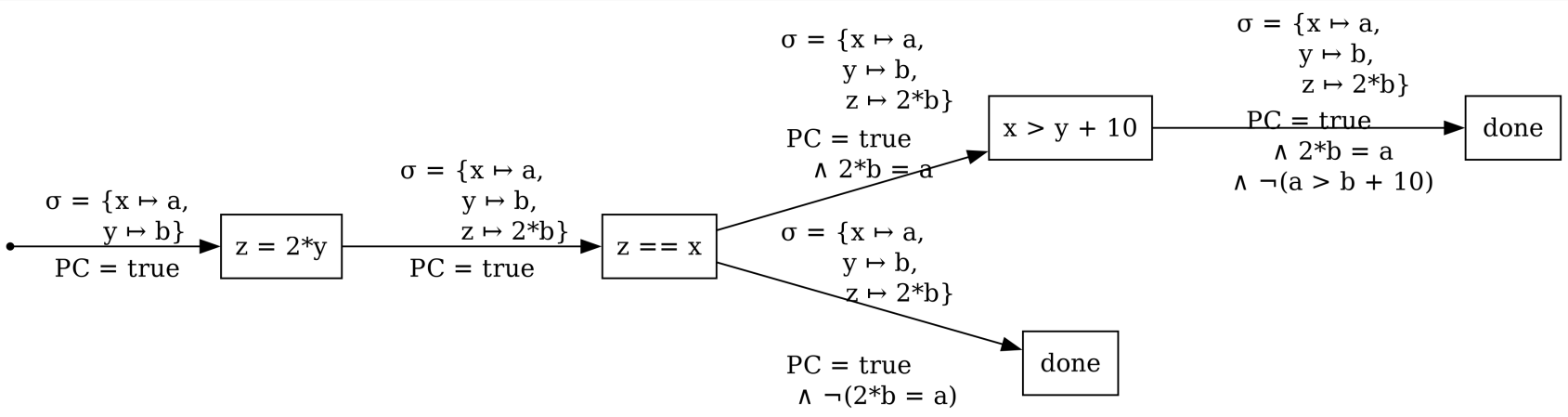
Exécution symbolique dynamique : exemple code/symex.c

Itération 2, avec `x = 0, y = 0`

```
void testme(int x, int y) {
    // σ = {x ↦ 0, y ↦ 0}, σ̂ = {x ↦ a, y ↦ b},  PC = true
    int z = 2*y;
    // σ = {x ↦ 0, y ↦ 0, z ↦ 0}, σ̂ = {x ↦ a, y ↦ b, z ↦ 2 * b}
    if (z == x) {
        // PC = true ∧ 2 * b = a
        if (x > y + 10) {
            abort();
        } else {
            // PC = true ∧ 2 * b = a ∧ ¬(a > b + 10)
        }
    } else {
    }
}
```

Sélection d'entrée pour l'itération suivante

symex.c



Avec `x = 0`, `y = 0`, on explore le chemin :

$$true \wedge 2 * b = a \wedge \neg(a > b + 10)$$

Sélection d'entrée pour l'itération suivante

symex.c

Il nous reste à explorer le chemin :

$$true \wedge 2 * b = a \wedge a > b + 10$$

```
$ z3 symex-concolic2.z3
sat
(
  (define-fun b () Int
    11)
  (define-fun a () Int
    22)
)
```

Code hors de notre portée

code/symex-unknown.c

```c
void testme(int x, int y) {
    // Contrainte non-linéaire, pas supportée par certains solveurs
    int z = (y * y) % 50;
    if (z == x) {
        if (x > y + 10) {
            assert(0);
        } else {

        }
    } else {
```

Code hors de notre portée

code/symex-unknown.c

Itération 1 avec `x = 22, y = 7`

```
void testme(int x, int y) {
    // σ = {x ↦ 22, y ↦ 7}, σ̂ = {x ↦ a, y ↦ b}, PC = true
    int z = (y * y) % 50;
    // σ = {x ↦ 22, y ↦ 7, z ↦ 49}, σ̂ = {x ↦ a, y ↦ b, z ↦ (b * b) mod 50}
    if (z == x) {
        if (x > y + 10) {
            abort();
        } else {
        }
    } else {
        // PC = true ∧ ¬((b * b) mod 50 = a)
    }
}
```

Code hors de notre portée

code/symex-unknown.c

On a exploré le chemin :

$$\mathit{true} \wedge \neg((b * b) \ \mathit{mod}\ 50 = a)$$

On souhaite résoudre :

$$\mathit{true} \wedge (b * b) \ \mathit{mod}\ 50 = a$$

Mais notre solveur ne supporte pas *mod* !

Solution : on remplace $(b * b)$ *mod* 50 par la valeur observée :

$$\mathit{true} \wedge 49 = a$$

On a donc `x = 49`, `y = 7`

(On garde l'ancienne valeur de `y` pour s'assurer de rester le plus possible le long du chemin précédent)

Code hors de notre portée

code/symex-unknown.c

```
void testme(int x, int y) {
    // σ = {x ↦ 49, y ↦ 7}, σ̂ = {x ↦ a, y ↦ b},  PC = true
    int z = (y * y) % 50;
    // σ = {x ↦ 49, y ↦ 7, z ↦ 49}, σ̂ = {x ↦ a, y ↦ b, z ↦ (b * b) mod 50}
    if (z == x) {
        // PC = true ∧ (b * b) mod 50 = a
        if (x > y + 10) {
            // PC = true ∧ (b * b) mod 50 = a ∧ a > b + 10
            abort();
        } else {
        }
    } else {
    }
}
```

Code hors de notre portée

code/symex-unknown.c

On a exploré le chemin :

$$\mathit{true} \wedge (b * b) \ \mathit{mod}\ 50 = a \wedge a > b + 10$$

On veut explorer le chemin restant :

$$\mathit{true} \wedge (b * b) \ \mathit{mod}\ 50 = a \wedge \neg(a > b + 10)$$

Avec $(b * b) \ \mathit{mod}\ 50 = 49$ (information dynamique), donc :

$$\mathit{true} \wedge 49 = a \wedge \neg(a > b + 10)$$

Code hors de notre portée

code/symex-unknown.c

$$\mathit{true} \wedge 49 = a \wedge a > b + 10$$

Z3 nous donne `a = 49`, `b = 38`

Code hors de notre portée

code/symex-unknown.c

```
void testme(int x, int y) {
    // σ = {x ↦ 49, y ↦ 38}, σ̂ = {x ↦ a, y ↦ b},  PC = true
    int z = (y * y) % 50;
    // σ = {x ↦ 49, y ↦ 38, z ↦ 49 }, σ̂ = {x ↦ a, y ↦ b, z ↦ (b * b) mod 50
    if (z == x) {
        if (x > y + 10) {
            abort();
        } else {
        }
    } else {
        // PC = true ∧ (b * b) mod 50 ≠ a)
    }
}
```

Code hors de notre portée

 `code/symex-unknown.c`

Cela ne permet pas d'atteindre le dernier chemin : on a perdu l'information que `(y * y) % 50 = x`.

Mais on a pu aller plus loin que prévu grâce à l'information de l'exécution concrète.

Stratégie d'exploration

On a fait le choix de faire la négation de la dernière clause à chaque itération.

C'est une **stratégie d'exploration**.

Stratégies possibles :

- DFS : léger sur la mémoire, mais peut rester bloquer dans des boucles ou appels récursifs.
- BFS : lourd sur la mémoire, mais ne reste pas bloqué
- Aléatoire
- Basé sur une notion de priorité, pour, par exemple :
  - maximiser la couverture
  - se rapprocher d'état sensibles

Utilisation réelle : Microsoft

Godefroid., Levin, & Molnar. (2012). SAGE: Whitebox Fuzzing for Security Testing: SAGE has had a remarkable impact at Microsoft. Queue.

Utilisé par exemple par Microsoft dans *SAGE* :

- exécuté après toutes les autres approches pour faire du *whitebox fuzzing*
- trouvé un tiers des bugs dans Windows 7
- tourne 24/7 sur 100+ machines

Utilisation réelle : DARPA Cyber Grand Challenge

En 2016, le DARPA (*Defense Advanced Research Projects Agency*) a mis en place un challenge :

- machine contre machine
- plusieurs tâches :
  - découverte automatisée de vulnérabilités
  - patch automatisé de binaires
  - génération automatisée d'exploits
  - implémentation de stratégies de sécurité

Les meilleurs finalistes utilisaient tous une forme de *whitebox fuzzing* avec résolution de contrainte.

Avgerinos et al. (2018). The Mayhem Cyber Reasoning System. IEEE Security & Privacy.

Outil récent : SymCC

Poeplau, & Francillon. (2020). Symbolic Execution with SymCC: Don't Interpret, Compile!. Usenix Security.

Observation :

- la plupart des moteurs d'exécution symbolique traduisent le binaire en IR qui est exécuté symboliquement
- cette exécution est similaire à un interpréteur et induit un ralentissement

Concept essentiel du papier :

- on peut embarquer le moteur symbolique dans le programme
- via de l'instrumentation au temps de compilation
- l'instrumentation est faite sur l'IR LLVM pour supporter plusieurs langages

SymCC : instrumentation

```
$ clang -O2 -S -emit-llvm symex.c -o symex.ll \
    -fno-discard-value-names
$ ./symcc -O2 -S -emit-llvm symex.c -o symex.symcc.ll \
    -fno-discard-value-names
$ diff --side-by-side symex.ll symex.symcc.ll
```

SymCC : Implémentation

```
$ git clone https://github.com/eurecom-s3/symcc
```

Structure :

- `compiler/` : une passe LLVM qui instrumente l'IR
  - `compiler/Pass.{cpp,h}` : la passe en elle-même
  - `compiler/Main.cpp` : enregistre la passe pour LLVM
  - `compiler/Runtime.{cpp,h}` : les symboles des fonctions ajoutées à l'instrumentation
  - `compiler/Symbolizer.{cpp,h}` : gère l'ajout de variables symboliques et la génération de contraintes
- `runtime/` : la bibliothèque utilisée par la partie instrumentée
  - `runtime/simple_backend/` : backend qui calcule l'entrée suivante et l'affiche
  - `runtime/qsym_backend/` : backend qui utilise *QSym* pour gérer l'exploration

Utilisation de SymCC

- Par défaut, seulement l'entrée lue depuis `stdin` est traitée symboliquement
- Exécuter le programme donne toutes les possibilités de divergences rencontrées sur une exécution
- Il faut faire tourner plusieurs fois avec différentes entrées : `util/pure_concolic_execution.sh`
- Ou combiner avec un fuzzer: `docs/Fuzzing.txt`

Exercice

Soit le programme suivant :

```
void testme(int x, int y) {
    if (x > y)
        if (x * y == 1452)
            if (x >> 1 == 5)
                y = y / (x - 10);
}
```

- Faire une exécution concolique à la main de la fonction `testme` en traitant ses arguments comme symbolique. Vous pouvez utiliser un solveur SMT pour résoudre les contraintes.
- Ce programme contient-il une erreur ?
- Exécuter SymCC sur ce programme.

Exercice

Exercice : faire tourner SymCC sur `code/password.c`

```c
int main(int argc, char *argv[]) {
    char input[15];
    if (read(STDIN_FILENO, input, sizeof(input)) != sizeof(input)) {
        fprintf(stderr, "Failed to read the input\n");
        return -1;
    }
    if (input[5] == input[7] && input[9] == input[5] &&
        input[2] == 'o' && //
        input[8] == input[6] + 1 &&
        input[11] == 121 &&
        input[sizeof(input)-1] == input[sizeof(input)-3] &&
        input[14] == 's' &&
        input[5] == 0x61 &&
        input[0] == input[1]-2 &&
        input[input[7] - input[9]] == 'p' &&
        input[13] == 'i'&&
        input[4] == input[1] &&
        input[6] == input[5] + 12 &&
        input[6] - 1 == input[10] &&
        input[3] == 'g' &&
        input[10] == 'l') {
        printf("OK!\n");
    }
    return 0;
}
```

Exécution symbolique : articles

Code source :

- Christakis, Müller, & Wüstholz. (2016). Guiding Dynamic Symbolic Execution toward Unverified Program Executions. ICSE.
- Busse, Nowack, & Cadar. (2020). Running Symbolic Execution Forever. ISSTA.

Exécution symbolique : articles

Binaires :

- Ming et al. (2015). LOOP: Logic-Oriented Opaque Predicate Detection in Obfuscated Binary Code, CCS.
- Huang, & White. (2022). Semantic-Aware Vulnerability Detection. CSR.
- Poeplau, & Francillon. (2021). SymQEMU: Compilation-Based Symbolic Execution for Binaries. NDSS.
- Yao et al. (2019). Identifying Privilege Separation Vulnerabilities in IoT Firmware with Symbolic Execution. ESORICS.

Exécution symbolique : articles

Web :

- Brown, Stefan, & Engler. (2020). Sys: A Static/Symbolic Tool for Finding Good Bugs in Good (Browser) Code. Usenix Security.
- Loring, Mitchell, & Kinder. (2019). Sound Regular Expression Semantics for Dynamic Symbolic Execution of JavaScript. PLDI.
- Azad et al. (2023). AnimateDead: Debloating Web Applications Using Concolic Execution. Usenix Security.

Exécution symbolique : outils

Beaucoup d'outils ne sont pas à code ouvert : DART (Bell Labs), SAGE (Microsoft), PEX (Microsoft), SymJS (Fujitsu), Mayhem (CMU).

Parmi les outils à code ouvert :

- SymCC : instrumentation dans le compilateur
- QSYM : exécution concolique dans le but d'être combiné avec un fuzzer hybride
- angr : plateforme d'analyse de binaires en Python
- Triton : exécution symbolique de binaires en Python
- Symbolic PathFinder : exécution symbolique de bytecode Java (un peu daté)
- KLEE : plateforme d'analyse symbolique construite sur LLVM
- Manticore : exécution symbolique pour binaires et *smart contracts*