Comparaison d'exécutables fondée sur des graphes

Halvar Flake, Rolf Rolles

Ruhr-Uni Bochum

SABRE Security GmbH

Introduction

- La non-divulgation des détails d'une vulnérabilité est chose commune dans l'industrie car les vendeurs des logiciels veulent que les clients aient du temps pour installer les patches (Comparer des objets exécutables est supposé être un problème asymétrique)
  - Changer le source et faire une recompilation est facile
  - Les vendeurs pensent que découvrir les changements entre les objets exécutables impliquent l'analyse complète des deux objets

Aujourd'hui, l'analyse complète de deux binaires n'est plus nécessaire pour les comparer.

Introduction (II)

- D’une perspective offensive, extraire la vulnérabilité exacte si on a que les objets exécutables avant/après le patch est bien précieux:
  - Beaucoup d’institutions prennent plus d’une semaine pour installer un patch
  - En Allemagne il est interdit aux institutions du monde de la finance d’installer un patch sans l’avoir testé pendant 7 jours
  - Pour compromettre un réseau important, l’attaquant peut attendre le patch corrigeant une nouvelle vulnérabilité pour la compromission initiale
  - La nécessité d’avoir un 0day est moins importante

Introduction (III)

- Du point de vue défensif, la capacité à faire une comparaison entres logiciels malicieux est utile:
  - L'analyse d'un nouveau malware est beaucoup plus rapide s'il existe une analyse d'un malware similaire
  - La classification automatique de logiciels malicieux pour former des "familles" est possible
  - Les similarités des logiciels malicieux peuvent être utilisées pour découvrir les collaborations entre différents auteurs

Comparaison du byte-code (I)

- L'approche directe pour faire une comparaison entre exécutables est la comparaison du byte-code
- Un changement minuscule dans le source peut entraîner une modification de beaucoup d'octets dans l'exécutable parce que le compilateur peut amplifier les changements:
  - Allocation de registres différents
  - Réorganisation des instructions
  - Inversion des instructions de branchement
  - Remplacement des instructions équivalentes
  - Changement du codage des instructions (SIB Byte)

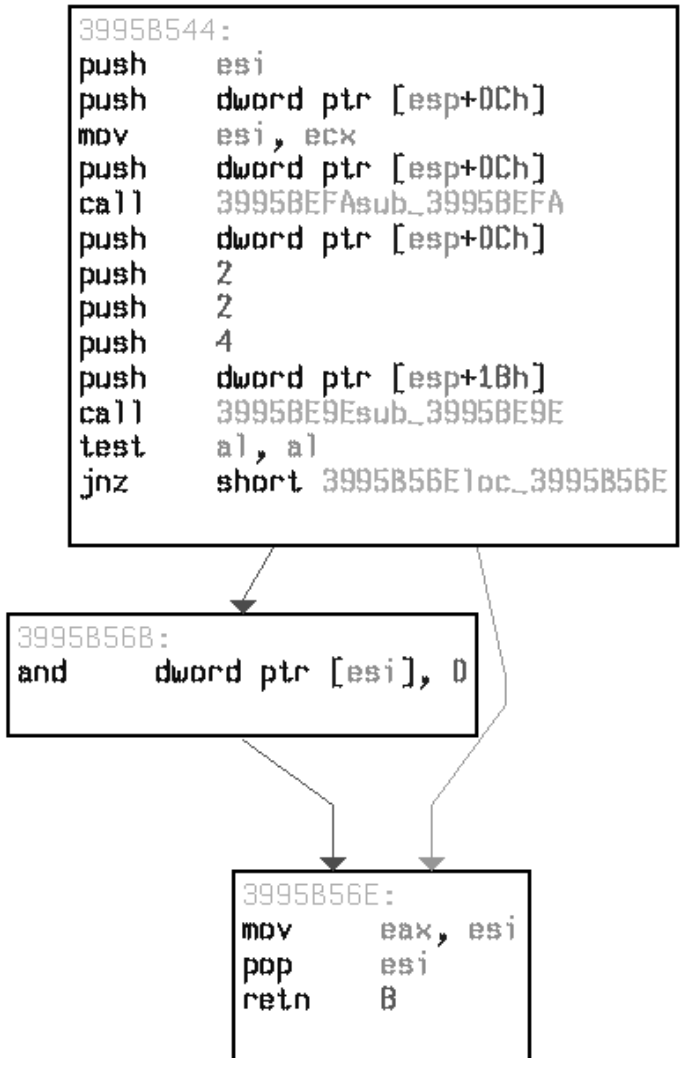
Comparaison du byte-code (II)

- En cas d'un changement de la configuration du compilateur, les changements peuvent être encore plus importants:
  - Omission du frame pointer
  - Omission d'instructions
  - Réorganisation agressive des blocs basiques (Profile-based optimisation)
- L'alignement des fonctions peut introduire un ordre différent de celles-ci dans l'exécutable
- Résumé: La comparaison du byte-code n'est pas une méthode satisfaisante

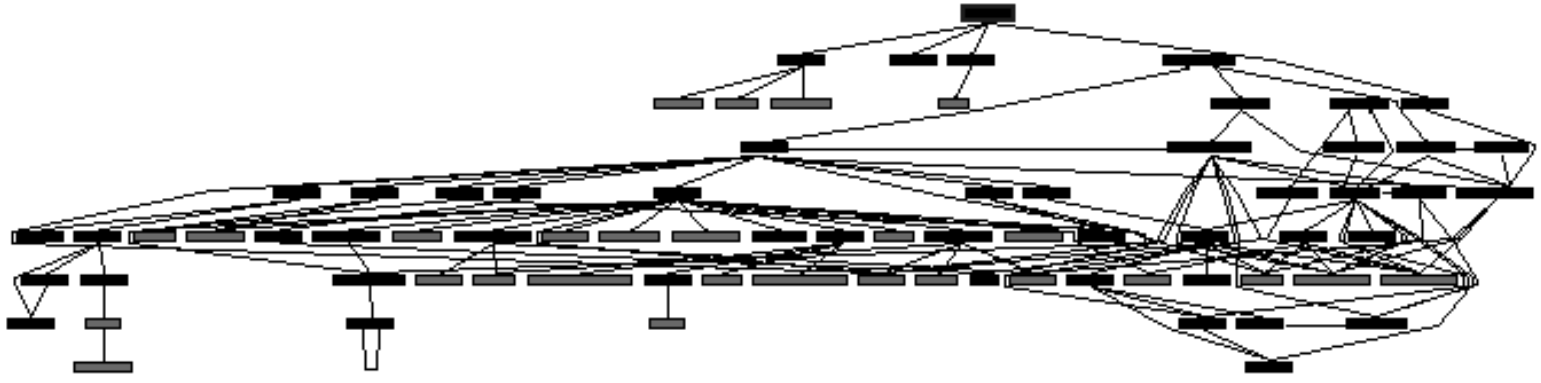
Une approche structurelle

- Une comparaison doit travailler sur les structures que le compilateur ne peut pas changer facilement
- La structure logique du logiciel (fonctions, conditions comme if, else) sont visibles dans l'exécutable $\Rightarrow$ On peut modéliser un logiciel avec un *graphe de graphes*
- Un logiciel consiste en un "callgraph":
  - Les sommets représentent les fonctions du logiciels
  - Les arcs représentent les relations "calls-to"
- Tous les sommets du "callgraph" sont aussi des graphes, les "cfgs" (control-flow-graph)
  - Les sommets représentent les blocs basiques
  - Les arcs représentent les relations "branches-to"

CFG/Flowgraph



Callgraph



L’invariance des graphes

Les graphes d’un exécutable sont plus ou moins invariants après une recompilation:

- Le ”callgraph” ne doit pas changer parce que la structure du source implique directement la structure du ”callgraph”
- Les ”cfgs” peuvent énormément changer, mais en essence la structure des fonctions sans boucle (seulement if, else) influe sur la structure et le nombre des blocs basiques

Notation

Le graphe G consiste en l'ensemble des sommets

$$G^n := \{G_1^n, \ldots, G_m^n\}$$

et en l'ensemble d'arcs

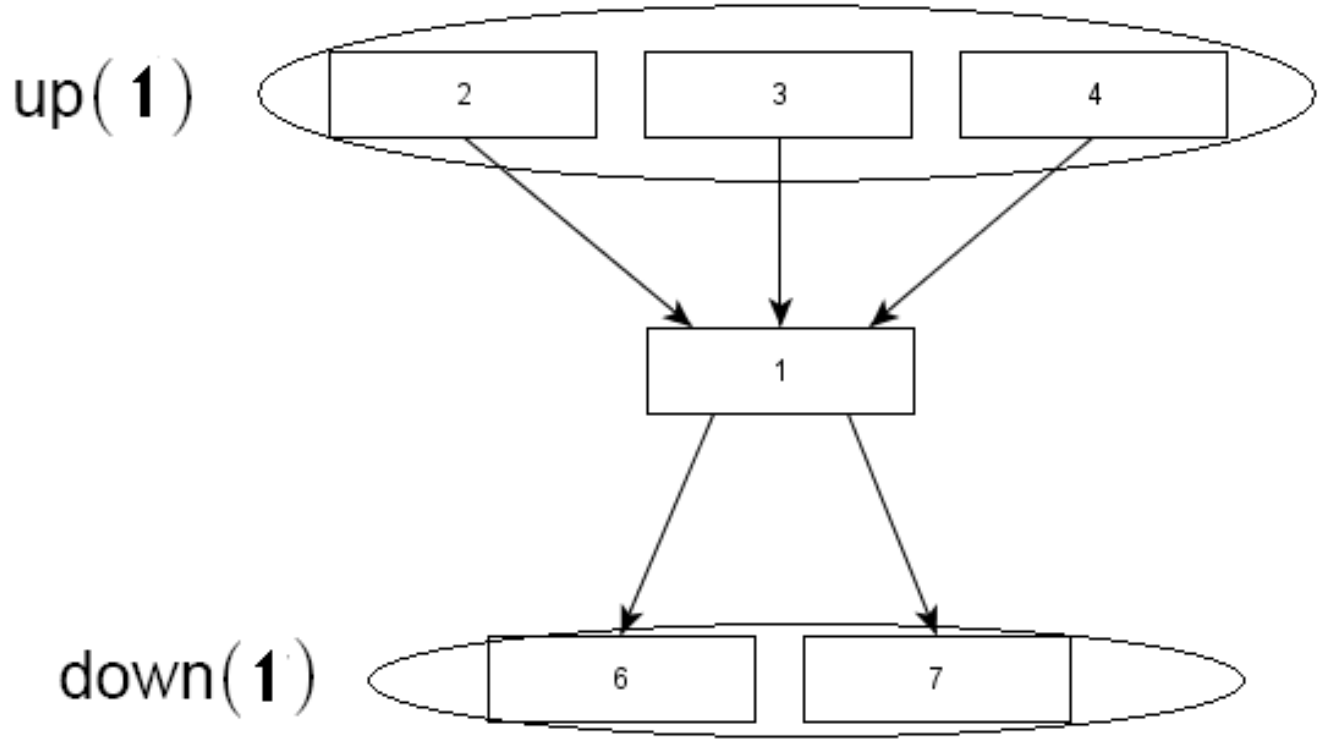
$$G^e := \{G_1^e, \ldots, G_k^e | G_i^e \in G^n \times G^n\}$$

Pour utilisation ultérieure on définit

$$\text{up} : G^n \rightarrow \mathfrak{P}(G^n) \quad , \quad \text{up}(G_i^n) \rightarrow \{G_j^n | (G_j^n, G_i^n) \in G^e\}$$
$$\text{down} : G^n \rightarrow \mathfrak{P}(G^n) \quad , \quad \text{down}(G_i^n) \rightarrow \{G_j^n | (G_i^n, G_j^n) \in G^e\}$$

qui associent un sommet G_i^n respectivement à l'ensemble de ses parents directs, et à l'ensemble de ses fils directs.

Up/Down



L’idée

L’idée: Faire une comparaison des exécutables fondée sur ces graphes:

1. Construire un isomorphisme entre les sommets des ”callgraphs” de deux exécutables
2. Avec un isomorphisme des fonctions, construire un isomorphisme entre les sommets des deux ”cfgs”
3. Avec un isomorphisme des blocs basiques, construire un isomorphisme entre les instructions des deux blocs basiques

Problème: Construire des isomorphismes de graphes est coûteux. On utilise donc des heuristiques stupides.

Définition: Sélecteur

Soient A^n, B^n ensembles de sommets des graphes. Le but de cet opérateur est de sélectionner le sommet $B' \subseteq B^n$ le plus similaire à un sommet donné de A^n (s'il n'est pas unique, on ne sélectionne rien).
Il est clair que la probabilité qu'un sélecteur choisisse un ensemble vide augmente avec la taille de l'ensemble des entrées.

Sélecteur pour le callgraph

On assigne un triplet à chaque sommet. Le triplet est constitué du nombre de blocs basiques,du nombre d'arcs et du nombre de fonctions appelées.

À partir d'un ensemble, le sélecteur retourne le sommet qui est à distance euclidienne minimale quand il est unique.

Sélecteur pour les *cfgs*

On utilise encore une fois un triplet avec les éléments suivants:

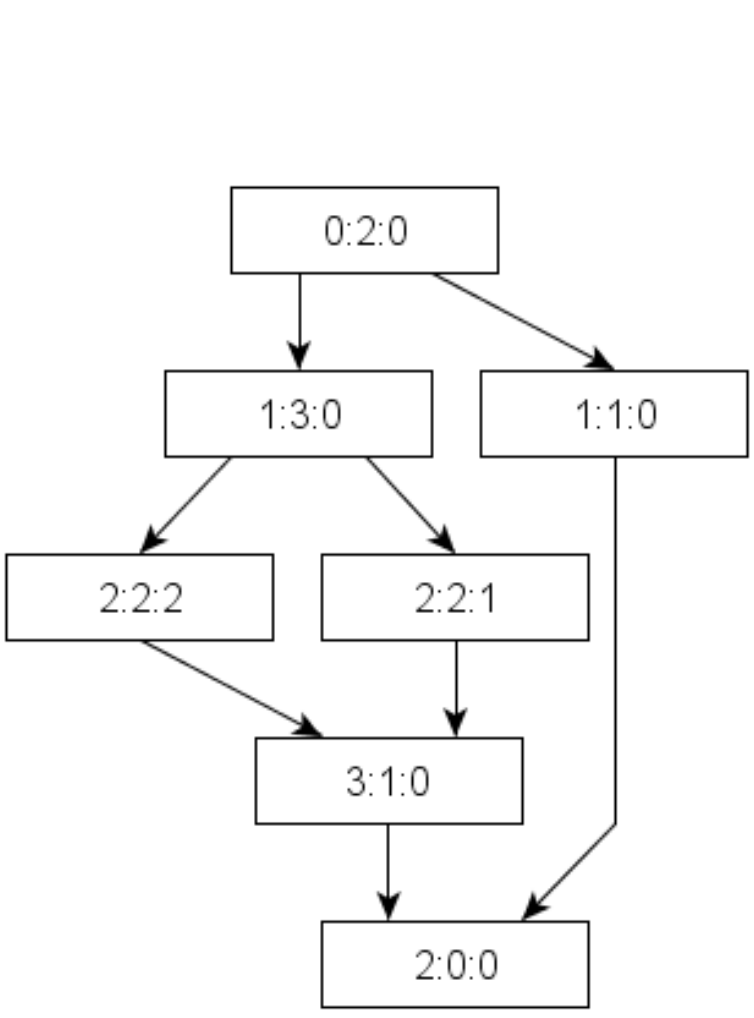
- nombre de ”sauts” depuis l’entrée du graphe
- nombre de ”sauts” jusqu’à une sortie du graphe
- nombre de fonctions appelées

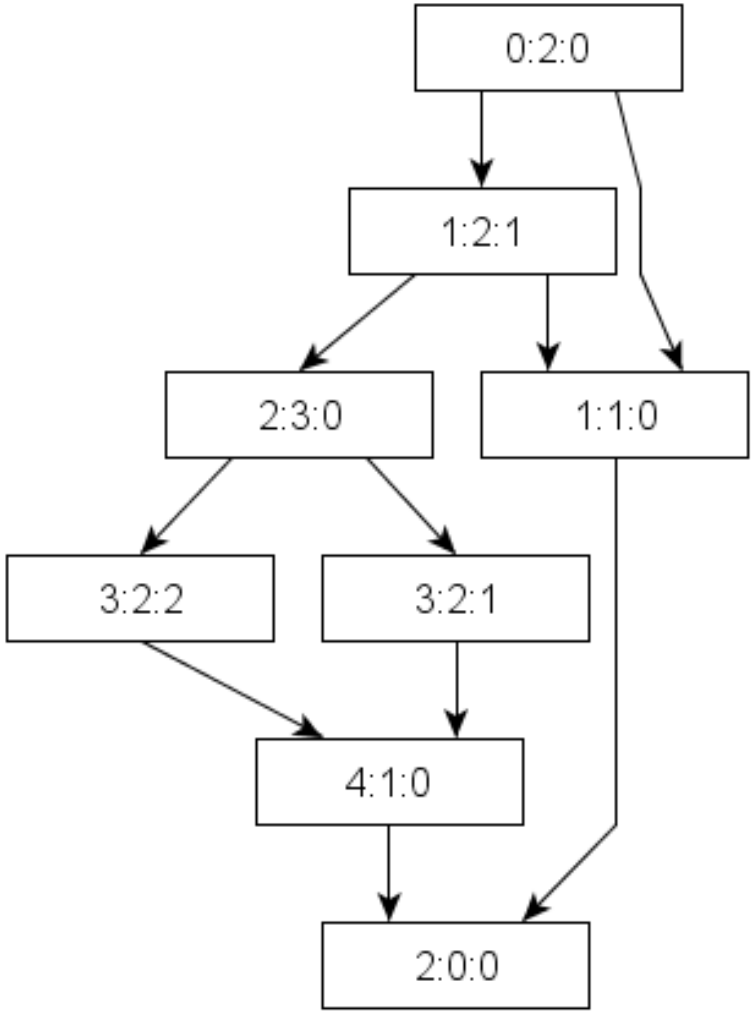
Si un sommet est inséré, tous les sommets dominés par celui-ci changent la signature. C’est la raison pour laquelle nous introduisons un δ:

$$s_c(x, A, \delta) :=$$

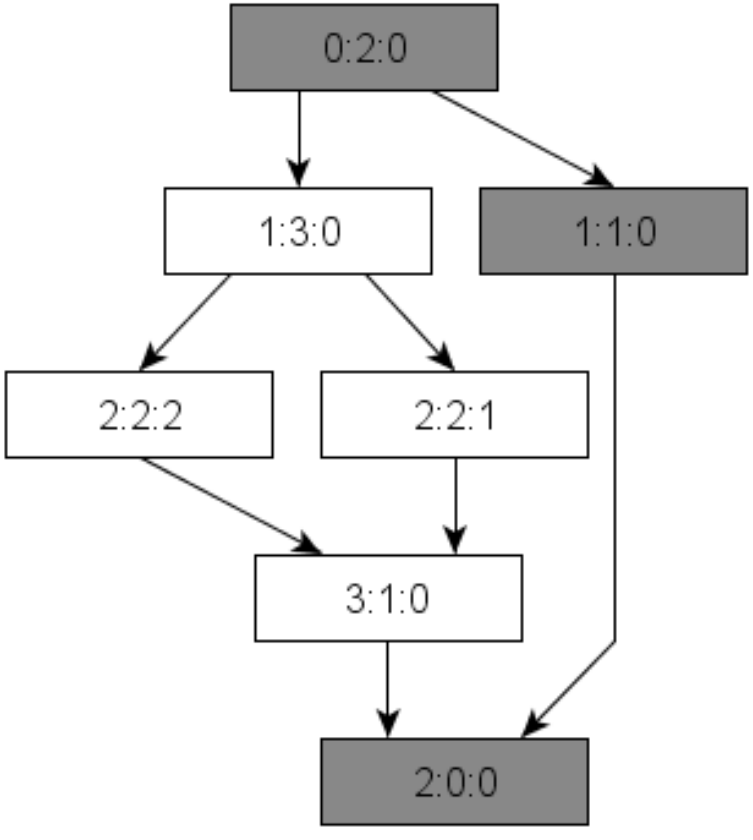
$$\begin{cases} a \text{ if } \exists_{a \in A} \forall_{b \in A, b \neq a} |x - (a + \delta)| < |x - (b + \delta)| \\ \emptyset \text{ else} \end{cases}$$

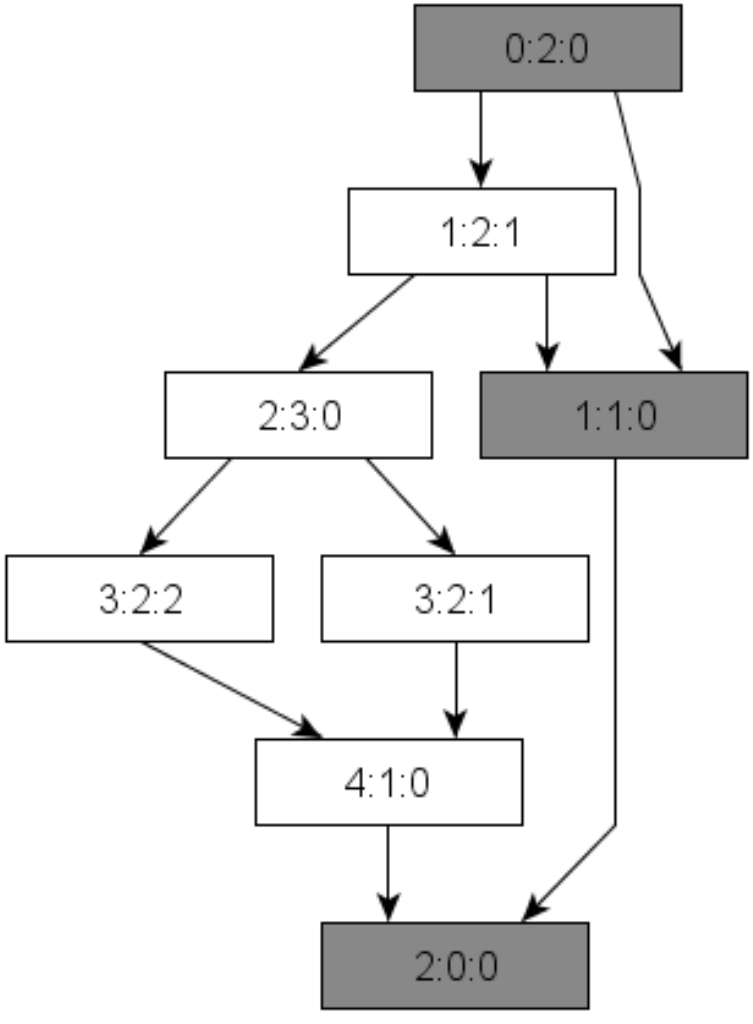
Sélecteur cfg



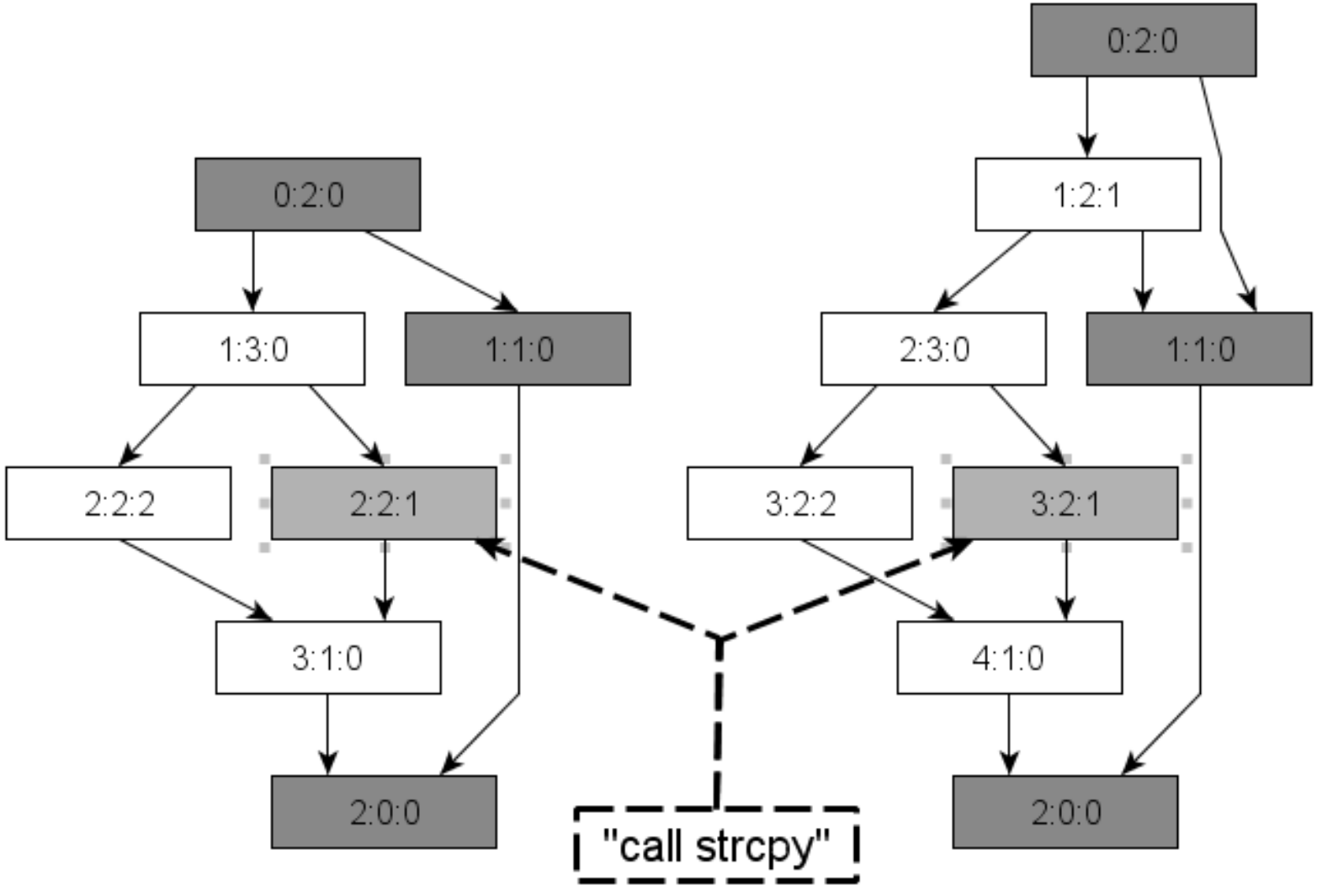


Sélecteur cfg

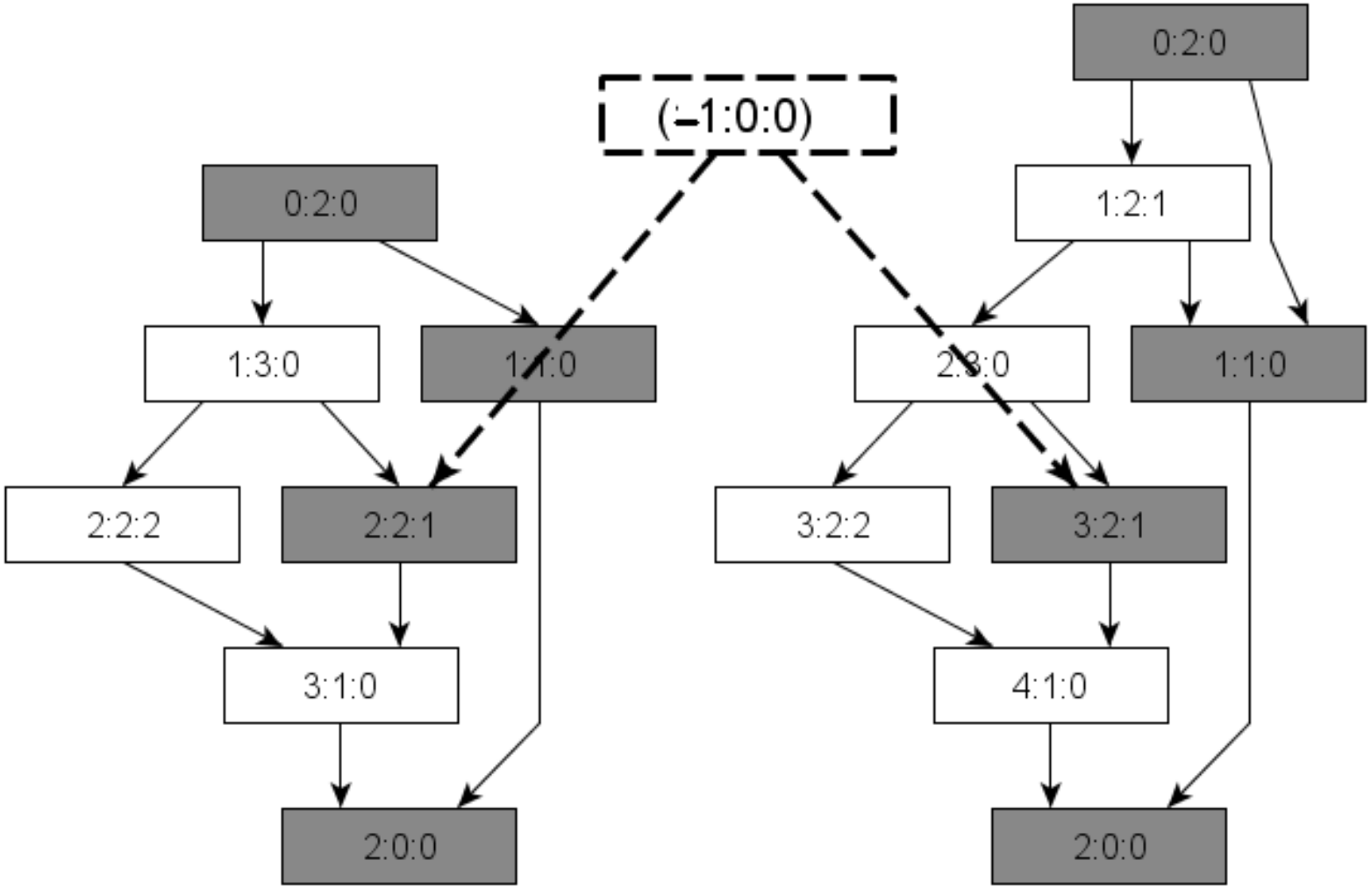




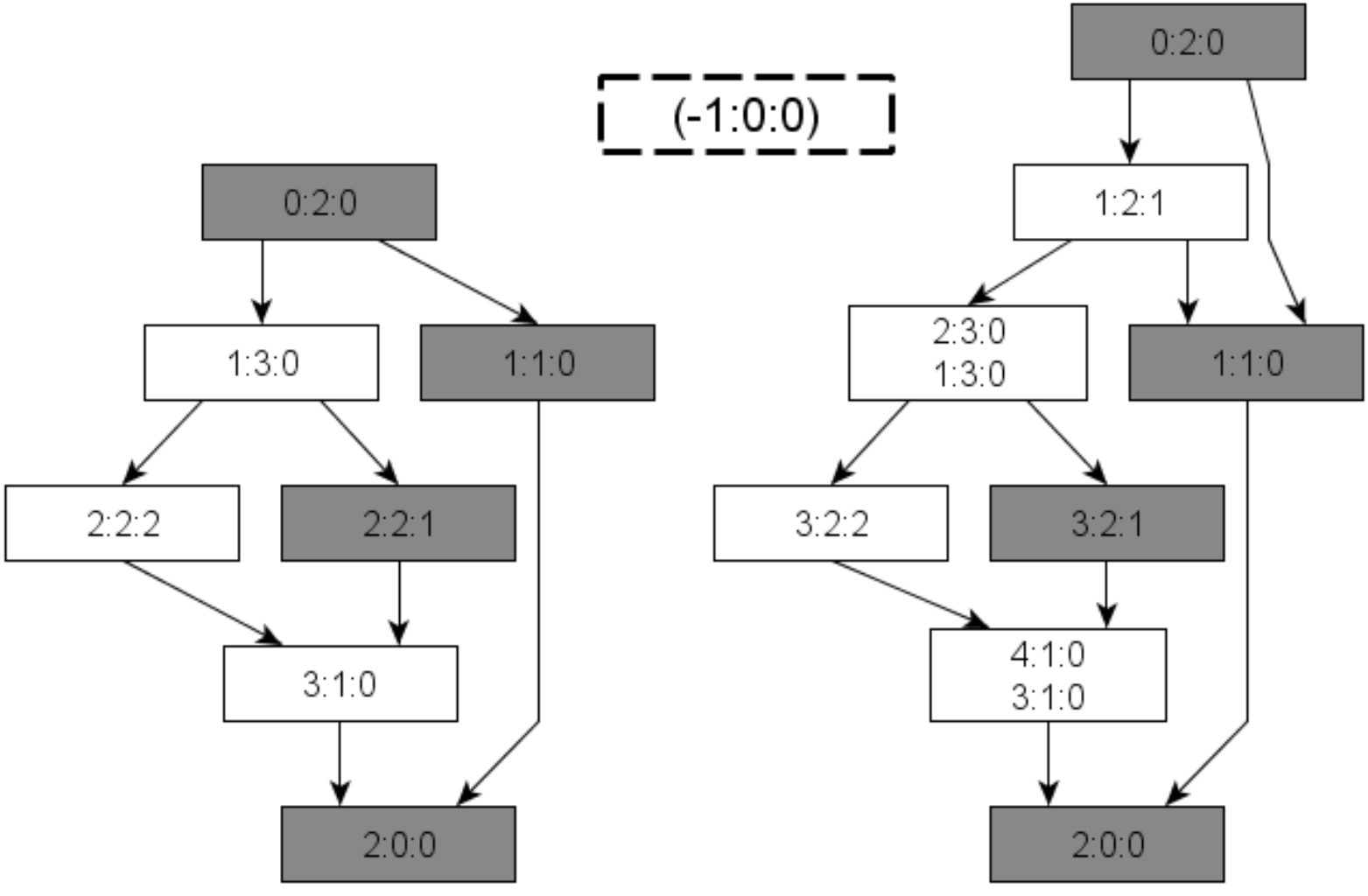
Sélecteur cfg



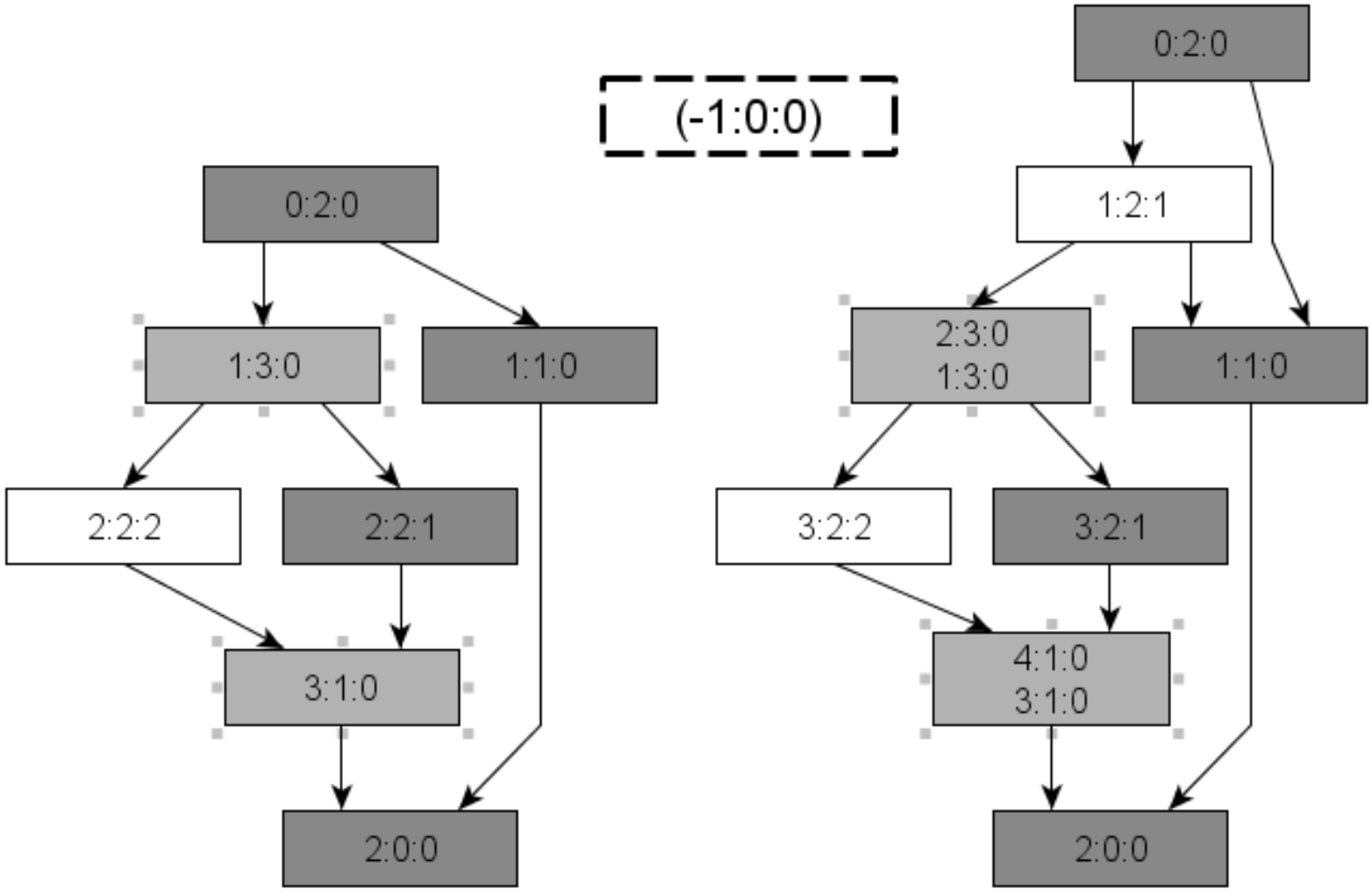
Sélecteur cfg



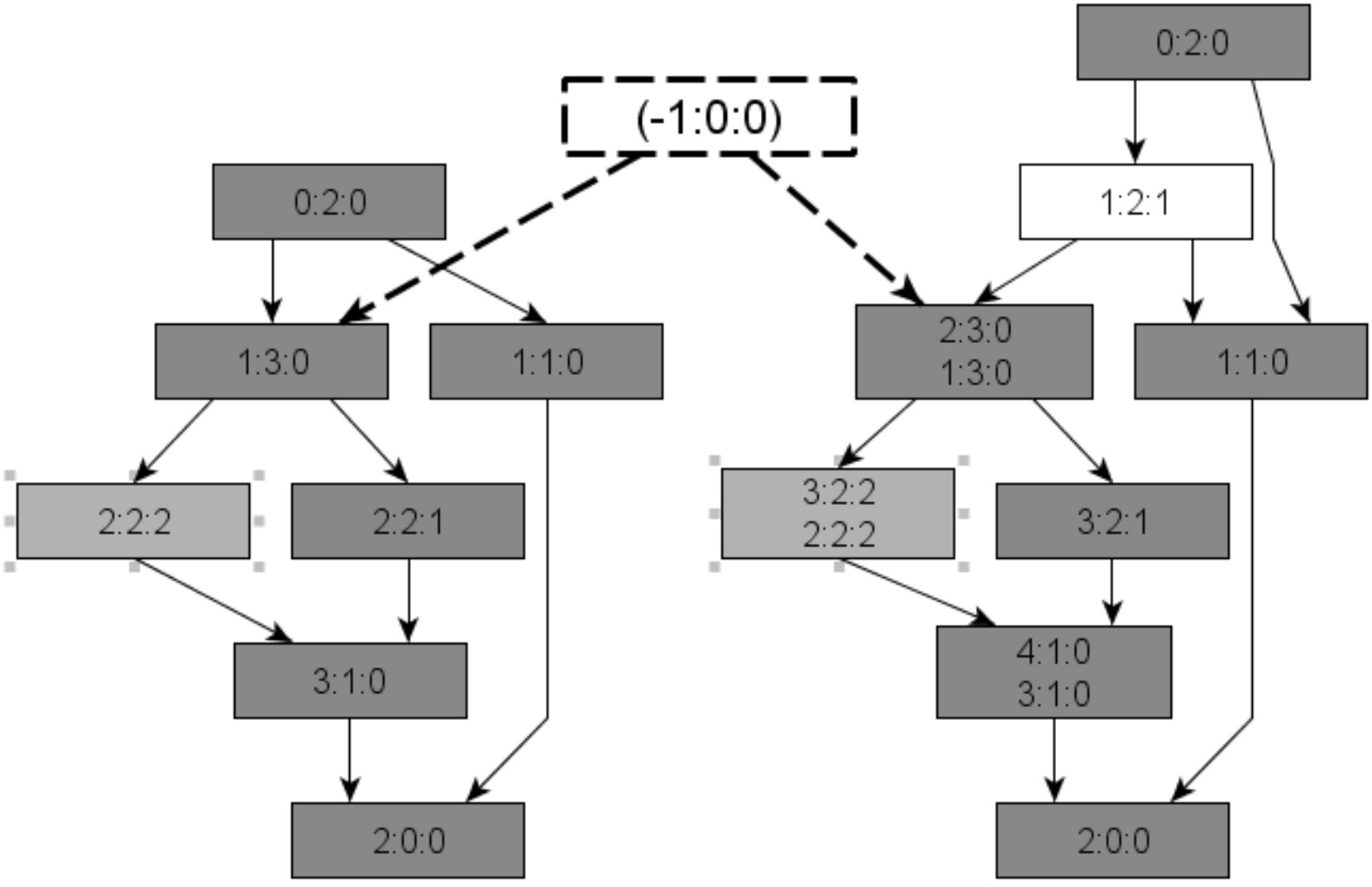
Sélecteur cfg



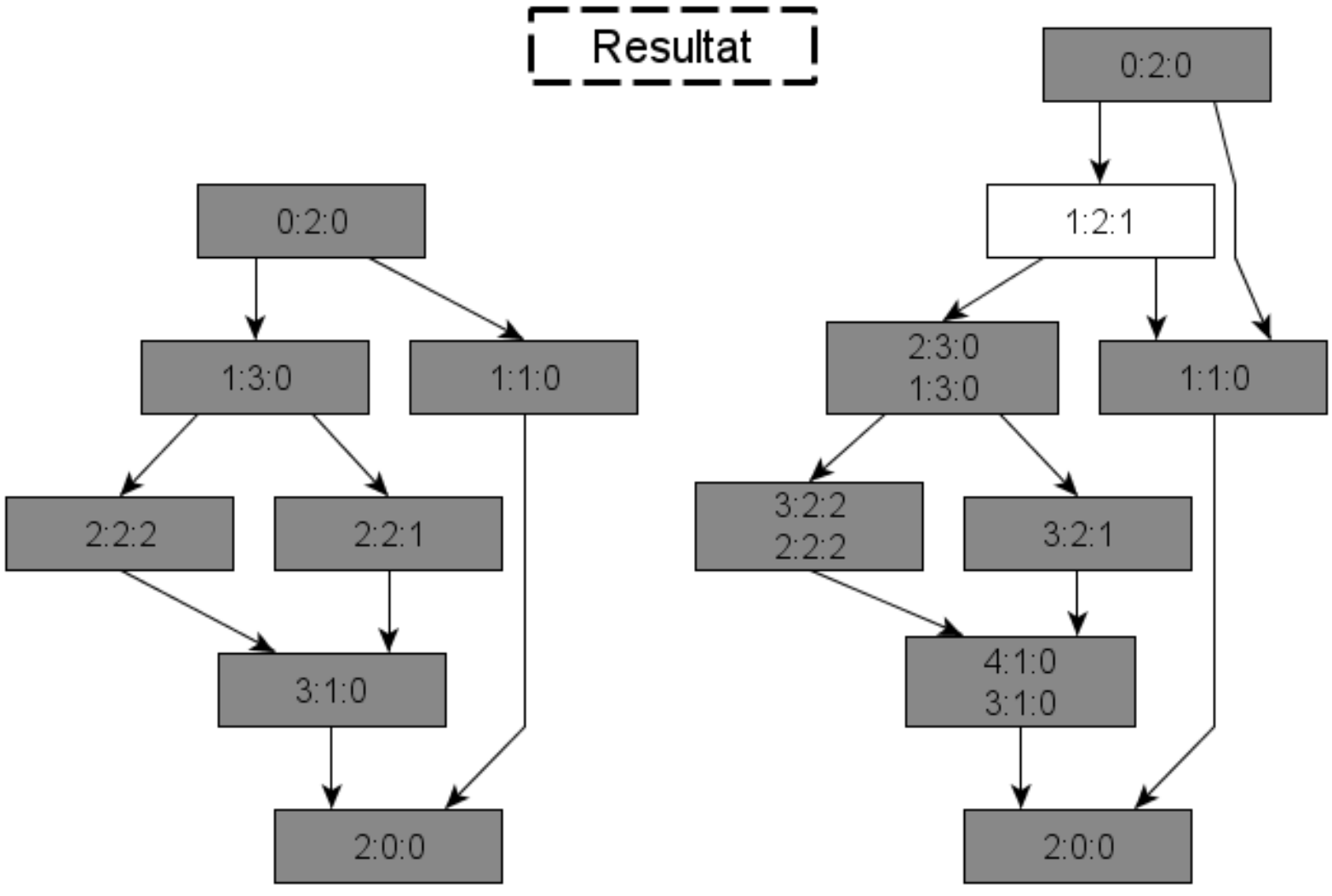
Sélecteur cfg



Sélecteur cfg



Sélecteur cfg



Définition: Propriétés

Une *propriété* π est définie par une association de deux graphes A et B d'après leur ensemble de sommets:

$$\pi(A, B) \rightarrow (A'^n, B'^n) \text{ avec } A'^n \subset A^n \text{ et } B'^n \subset B^n$$

Le but d'une telle association est de réduire la taille des ensembles utilisés par le sélecteur pour augmenter la probabilité que le sélecteur choisisse un résultat non vide.

Exemples de Propriétés

- *k-up / k-down*

$$\sharp(\text{up}(A_i^n)) = k \text{ et } \sharp(\text{up}(B_i^n)) = k$$

$$\sharp(\text{down}(A_i^n)) = k \text{ et } \sharp(\text{down}(B_i^n)) = k$$

- *Récursivité*

$$A_i^n \in \text{up}(A_i^n) \text{ et } B_i^n \in \text{up}(B_i^n)$$

- *Nom identique* (grâce aux symboles ou aux exports)
- *Références aux chaînes des caractères identiques*
- *SSP (”small primes product”) identiques*

Small-Prime-Product

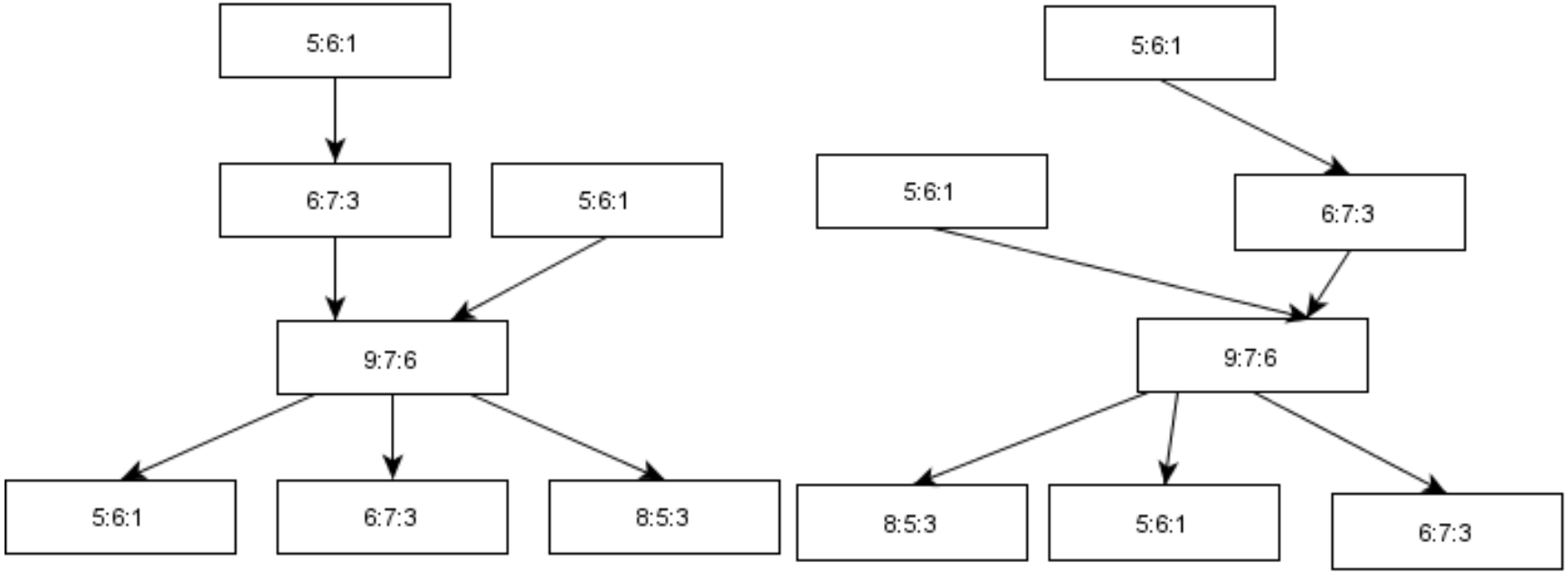
- Assigne à chaque instruction un petit entier premier unique
- Assigne le même petit entier aux instructions de branchement avec des conditions complémentaires (e.g. jz, jnz)
- Assigne l'entier 1 aux instructions de branchement inconditionnel
- Pour une suite d'instructions, on calcule le produit de tous les entiers de cette suite
- $\Rightarrow$ Résultat: L'unicité de la décomposition en facteurs premiers et la commutativité de la multiplication nous donne une signature pour une suite d'instructions qui est indépendante de l'ordre des instructions

Association Initiale

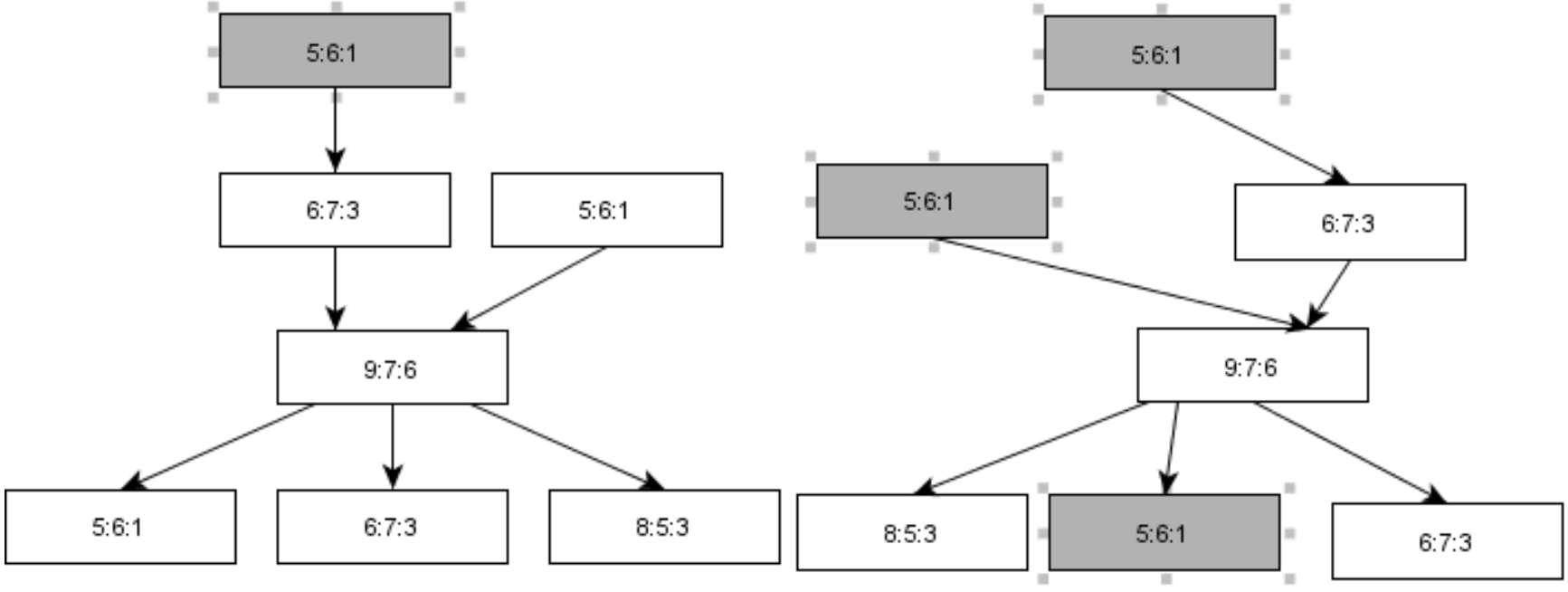
Avec un sélecteur s et un ensemble de propriétés $\Pi = \{\pi_1, \ldots, \pi_k\}$ on peut construire un isomorphisme initial:

for $\pi \in \Pi$ **do**
  $(K, L) \leftarrow \pi(A, B)$;
  for $x \in K$ **do**
    define $p_1(x) \rightarrow s(x, L)$
  end
end

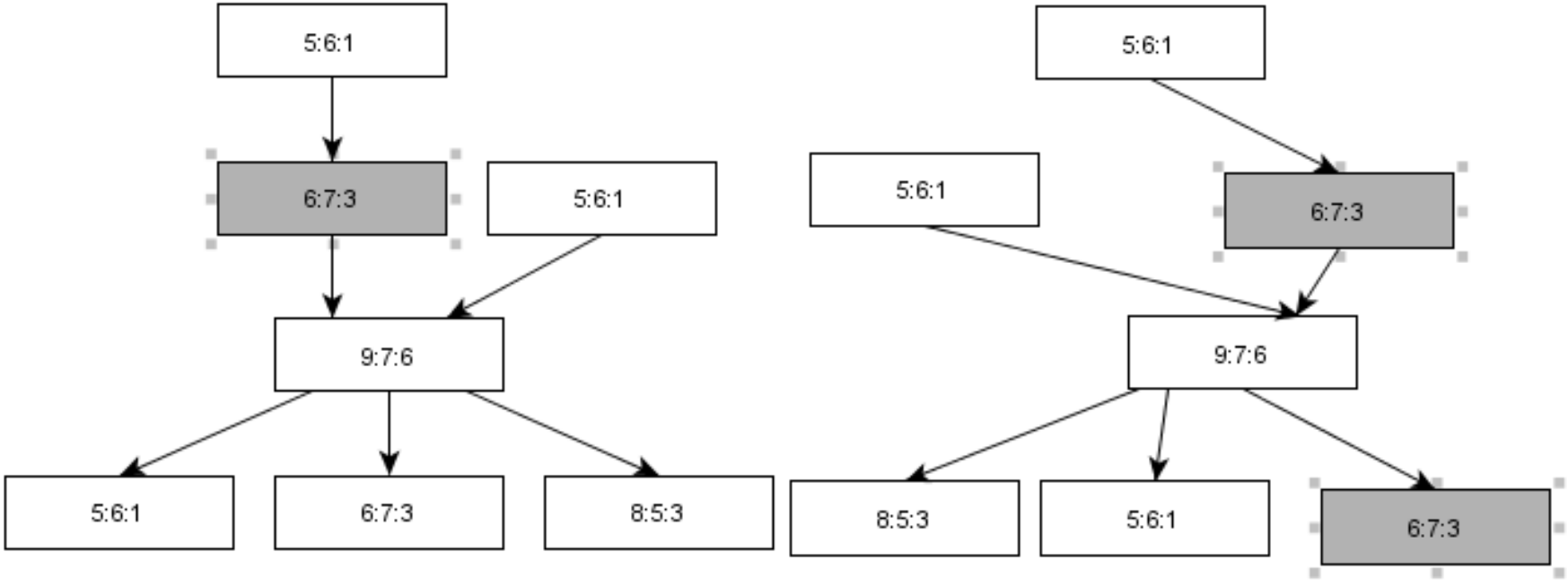
isomorphisme initial



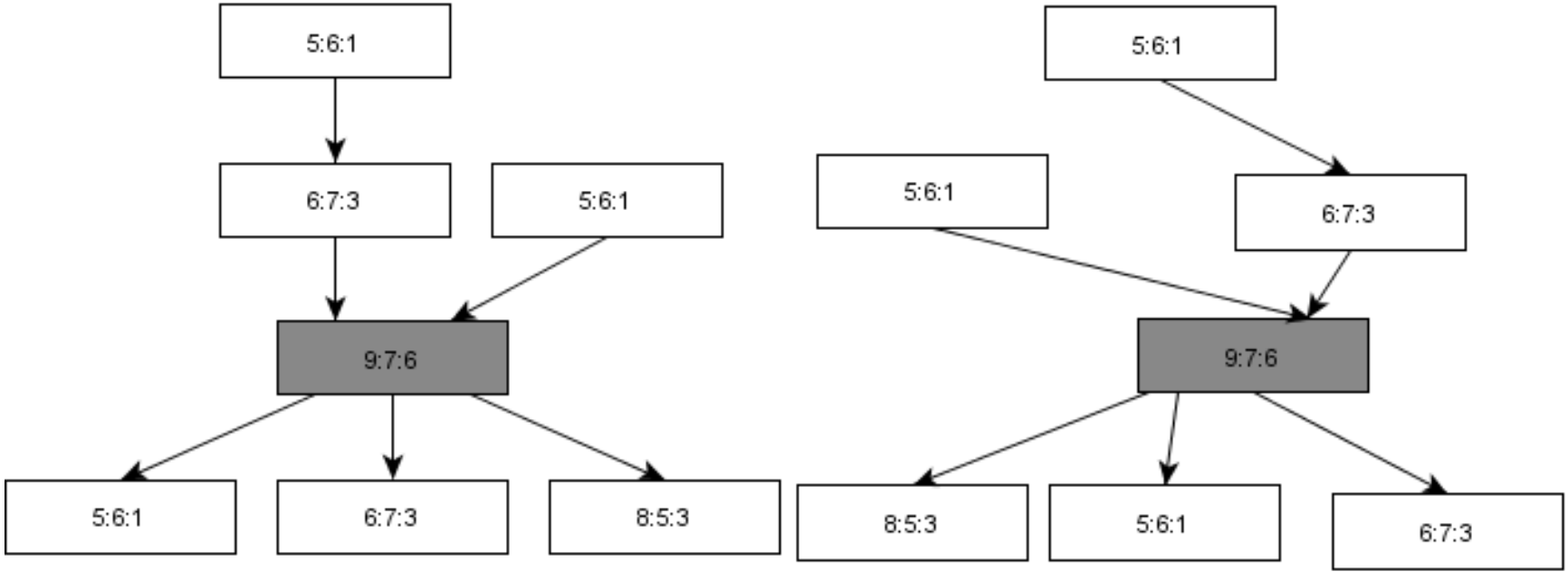
isomorphisme initial



isomorphisme initial



isomorphisme initial

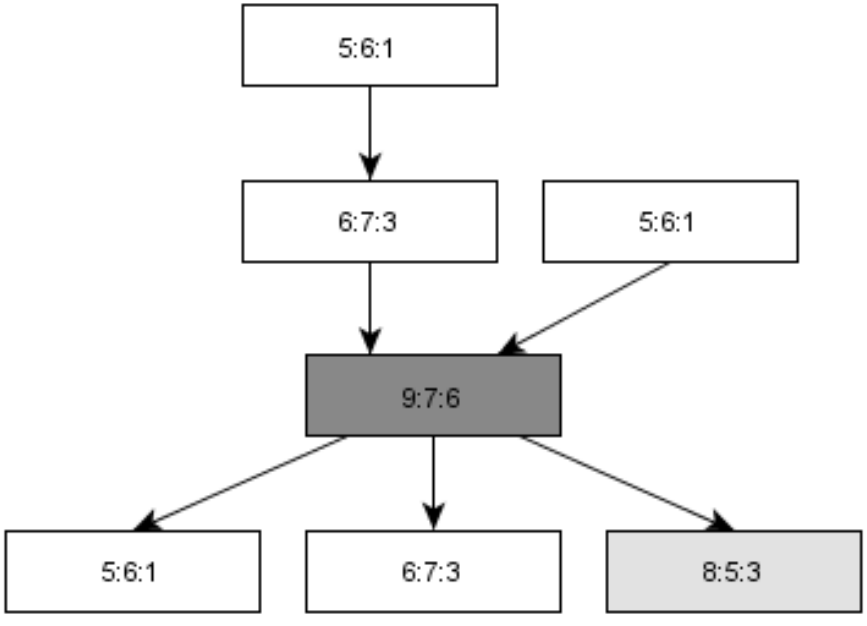


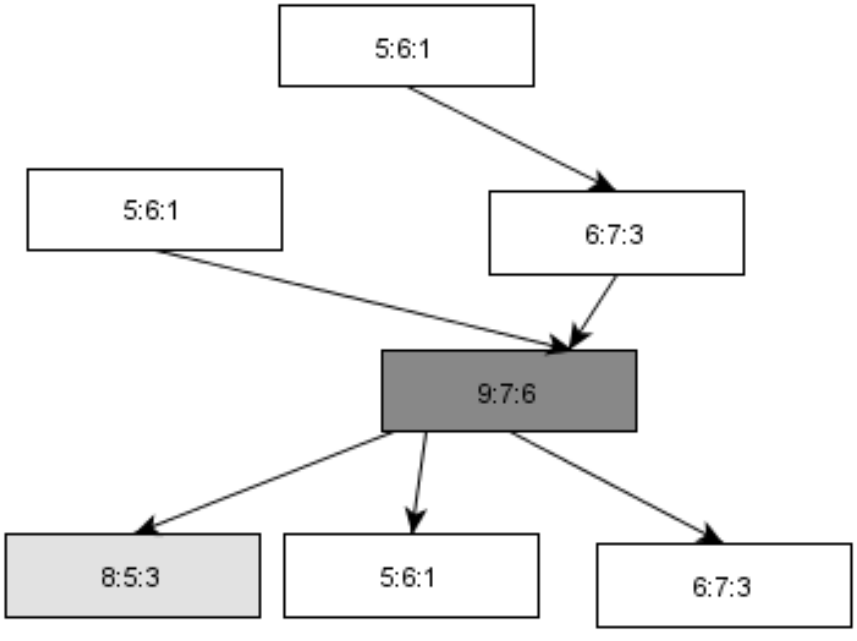
Points fixes et propagation (III)

On peut commencer à améliorer l'isomorphisme initial:

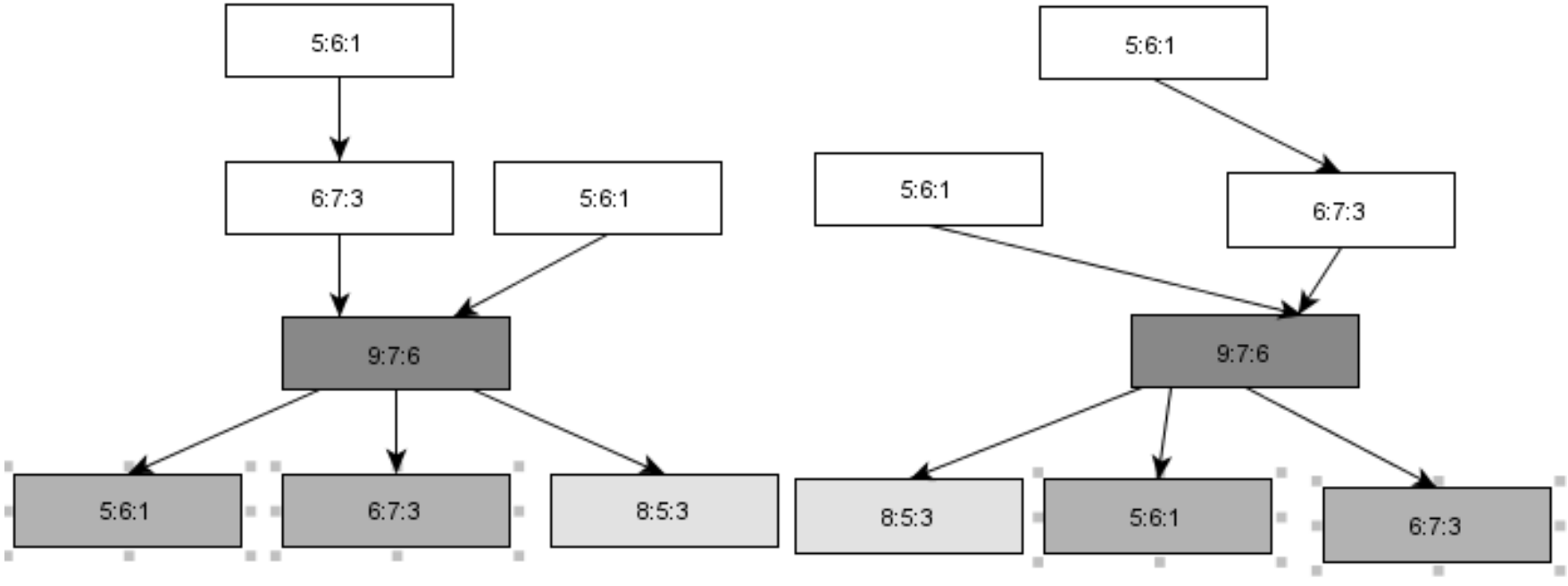
Input: p_{n-1}, s, A, B
Result: p_n
$S \leftarrow \{x \in A^n | p_{n-1}(x) \neq \emptyset\}$;
for $x \in S$ S **do**
  P $\leftarrow$ up(x)
  K $\leftarrow$ up$(p_{n-1}(x))$;
  for $y \in P$ **do**
    if $s(y, K) \neq \emptyset$ **then**
      define$p_n(y) \rightarrow s(y, K)$
    end
  end
end

Propagation

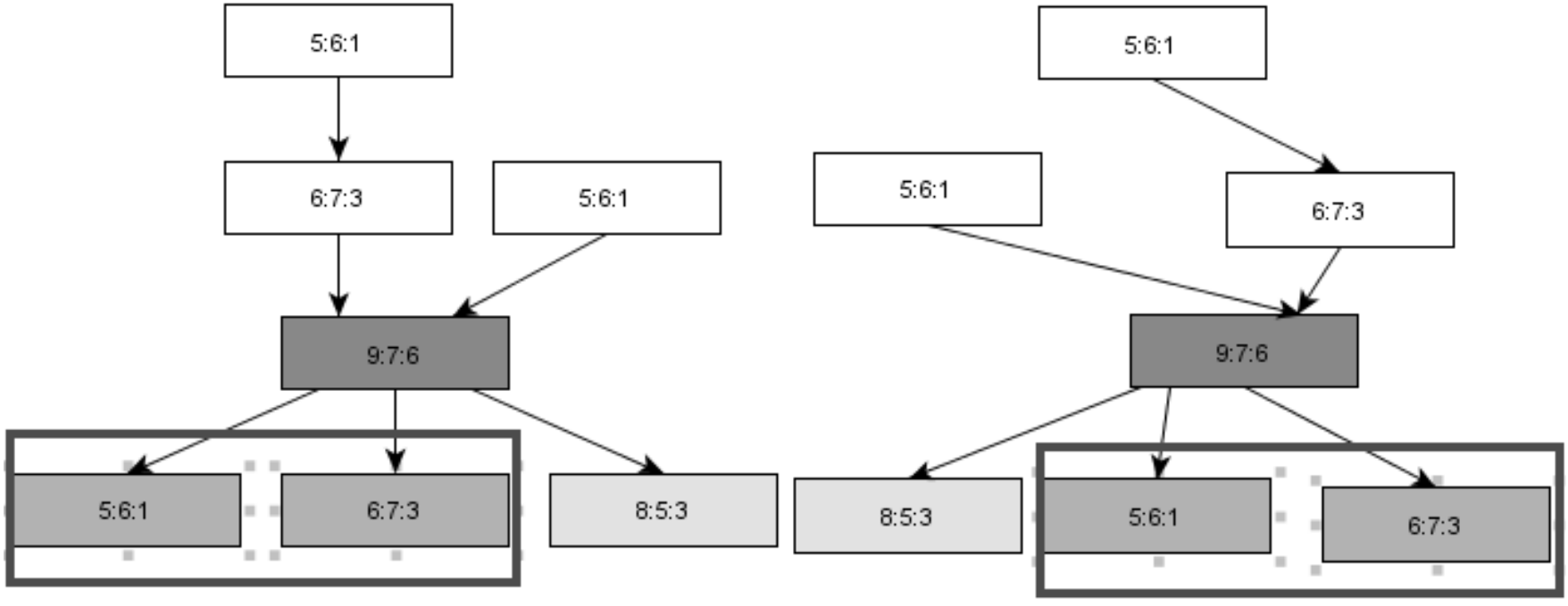




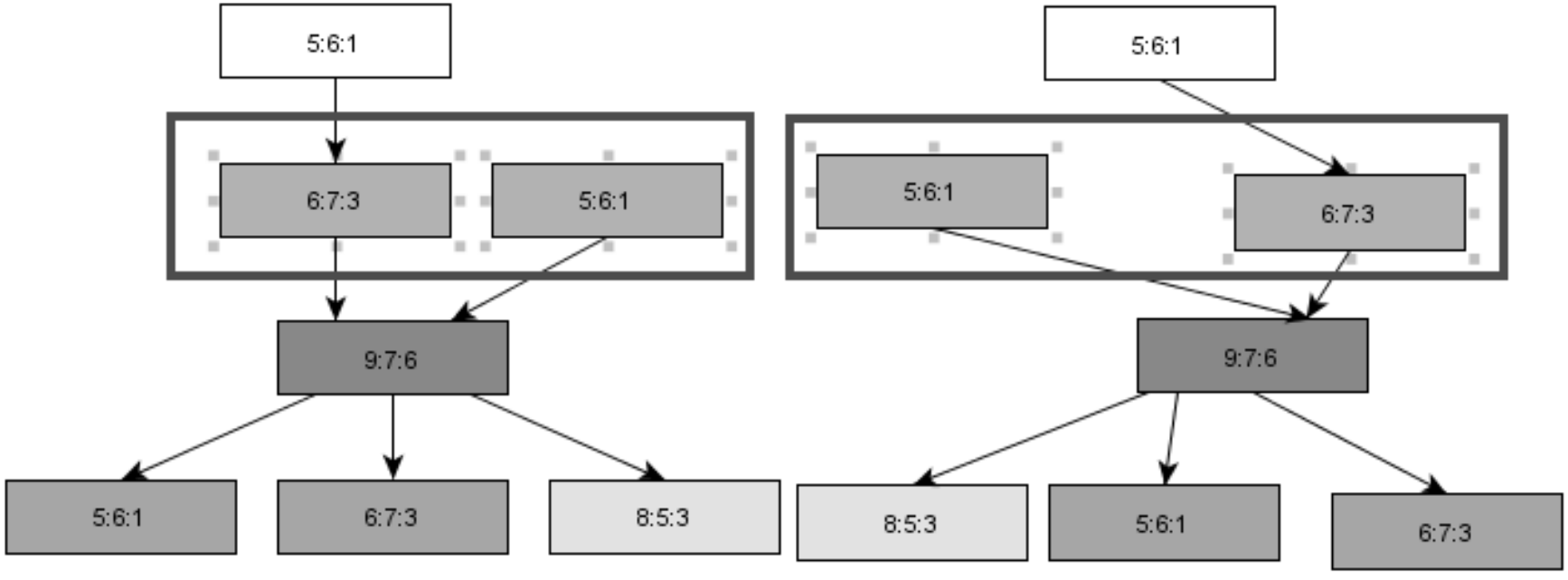
Propagation



Propagation



Propagation



Applications

- Analyse différentielle de patches
- Analyse de logiciels malicieux

Questions ?

halvar.flake@sabre-security.com
http://www.sabre-security.com