Université de Namur

INFOM227 - Program Analysis for Cybersecurity

Project 1 : Abstract Interpretation

Final Report

December 2024

BOUNCER Yassine
FUNDU Oliver
HARITANE Ryad

Table des matières

1 Introduction

Dans le cadre du cours *Program analysis for Cybersécurity* (INFOM227), ce rapport se concentre sur la mise en œuvre d'un outil d'analyse statique de programme. L'objectif principal de ce projet est d'étendre le langage de programmation Small pour prendre en charge les tableaux et développer un analyseur capable de détecter les erreurs courantes liées aux tableaux, telles que les accès invalides aux tableaux.
Le rapport couvre les points suivants :

- **Description Informelle de l'Analyseur** : Cette section fournit un aperçu des objectifs et des capacités de l'analyseur, y compris les types d'erreurs qu'il peut détecter et les métriques de performance qu'il rapporte.
- **Syntaxe et Sémantique Concrètes du Langage** : Ici, nous présentons la syntaxe de la grammaire étendue et la sémantique formelle pour le langage Small afin de prendre en charge les tableaux. Cela inclut les nouvelles règles de grammaire et les nœuds correspondants de l'arbre syntaxique abstrait (AST).
- **Domaine Abstrait et Lattice** : Cette section décrit le domaine abstrait et la structure de la lattice utilisée dans l'analyse. Elle explique comment la lattice est construite et comment elle permet de détecter les erreurs liées aux tableaux.
- **Fonctions d'Abstraction et de Concrétisation** : Nous définissons les fonctions d'abstraction et de concrétisation qui mappent les états de programme concrets aux états abstraits et vice versa. Ces fonctions sont cruciales pour la solidité et la complétude de l'analyse.
- **Fonctions de Flux** : Cette section détaille les fonctions de flux utilisées dans l'analyse. Nous fournissons des exemples de fonctions de flux intéressantes et discutons de leur rôle dans le processus d'analyse.
- **Solidité et Complétude** : Nous analysons la solidité et la complétude de l'analyse statique. Si possible, nous fournissons une preuve de la propriété de solidité locale pour l'une des fonctions de flux.
- **Cas d'Erreurs et d'Avertissements** : Cette section explique les différents cas d'erreurs et d'avertissements que l'analyseur peut détecter. Nous décrivons les conditions dans lesquelles ces cas se produisent et les types de messages retournés par l'analyseur.
- **Direction de l'Analyse** : Nous discutons si l'analyse est avant, arrière ou utilise une autre approche, et justifions la direction choisie.
- **Programme d'Entrée Exemple** : Cette section fournit un programme d'entrée exemple et explique, étape par étape, les calculs effectués par l'analyse en utilisant l'algorithme de liste de travail.
- **Procédure d'Exécution** : Nous décrivons la procédure d'exécution de l'analyseur, y compris toute configuration nécessaire et les commandes.

À travers ce rapport, nous visons à démontrer le processus d'extension d'un langage de programmation et de mise en œuvre d'un outil d'analyse statique pour améliorer la fiabilité et la sécurité des programmes.

2 Informal Description of the Analyzer

L'objectif de notre analyseur est d'étendre le langage de programmation Small pour inclure les tableaux et de mettre en œuvre un outil d'analyse statique de programme capable de détecter les problèmes liés à l'utilisation des tableaux. En appliquant l'interprétation abstraite, notre outil garantit que les programmes écrits dans le langage Small étendu adhèrent aux meilleures pratiques et évitent les erreurs liées aux tableaux (recherche par index négatif).

— **Analyse Statique de l'Utilisation des Tableaux** : L'analyseur utilise l'interprétation abstraite pour garantir des opérations sur les tableaux sûres et prévisibles. Les analyses clés incluent :
  — *Avertissements d'Indice Négatif* : Identification et signalement des cas où des indices négatifs pourraient être utilisés.
  — *Erreur d'Indice Négatif* : Identification et signalement des cas où des indices négatifs sont utilisés de manière certaine.

— **Environnement Abstrait et Lattice** : L'analyseur modélise l'état du programme en utilisant un environnement abstrait, en s'appuyant sur une lattice personnalisée pour les signes (*Négatif*, *Zéro*, *Positif*, *Inconnu*, *Bottom* et *Booléen*) pour suivre la plage de valeurs des variables et des indices de tableau.

En résumé, notre analyseur fournit un cadre pour détecter et résoudre les problèmes liés aux tableaux dans le langage de programmation Small étendu, améliorant ainsi la sécurité et la fiabilité du code grâce à des techniques d'analyse statique solides.

3 Concrete Syntax and Semantics of the Language

3.1 Extended Grammar Syntax

— **Déclaration (`declareStatement`) :**
  — Ancienne grammaire :

```
declareStatement: type IDENTIFIER SEMICOLON;
```

  — Nouvelle grammaire (extension pour les tableaux dynamiques) :

```
declareStatement: type IDENTIFIER (LBRACE NUMBER RBRACE)? SEMICOLON;
```

  - Permet de déclarer des tableaux dynamiques, e.g., `int tab{2};`.

— **Assignation (`assignStatement`) :**
  — Ancienne grammaire :

```
assignStatement: IDENTIFIER ASSIGN (expression
| LBRACE scope expression RBRACE)
SEMICOLON;
```

  — Nouvelle grammaire (extension pour les tableaux dynamiques) :

```
assignStatement: IDENTIFIER (LBRACE expression RBRACE)?
ASSIGN (expression | LBRACE scope expression RBRACE) SEMICOLON;
```

- Ajout de la gestion des assignations avec des tableaux, e.g., `tab{1} = 2;`.

— **Types (`type`) :**
  — Ancienne grammaire :
```
type: INT | BOOL;
```
  — Nouvelle grammaire (extension pour les types tableau) :
```
type: INT | BOOL | INT LBRACE RBRACE | BOOL LBRACE RBRACE;
```
  - Supporte les types tableau comme `int` ou `bool`.

— **Atomes (`atom`) :**
  — Ancienne grammaire :
```
atom: IDENTIFIER | NUMBER | TRUE | FALSE | LPAR expression RPAR;
```
  — Nouvelle grammaire (extension pour l'accès aux tableaux) :
```
atom: IDENTIFIER (LBRACE expression RBRACE)?
      | NUMBER | TRUE | FALSE | LPAR expression RPAR;
```
  - Ajout de la gestion de l'accès aux éléments des tableaux, e.g., `tab{1}`.

3.2 Formal Semantics

[Array declaration] $$\frac{x \notin dom(\sigma) \quad t \in \{Int, Bool\} \quad n > 0 \quad \sigma' = \sigma[x \mapsto \text{Array}(t,n)]}{(\text{array } x\{n\}, \Sigma \bullet \sigma) \rightsquigarrow \Sigma \bullet \sigma'}$$

[Array access] $$\frac{(e, \Sigma \bullet \sigma) \rightsquigarrow i \quad i \in \mathbb{Z} \quad \sigma(x) = \text{Array}(t,n) \quad 0 \leq i < n \quad v = x\{i\}}{(x\{e\}, \Sigma \bullet \sigma) \rightsquigarrow v}$$

[Array assignment] $$\frac{(e_1, \Sigma \bullet \sigma) \rightsquigarrow i \quad (e_2, \Sigma \bullet \sigma) \rightsquigarrow v \quad i \in \mathbb{Z} \quad v \in t \quad \sigma(x) = \text{Array}(t,n) \quad 0 \leq i < n}{(x[e_1] = e_2, \Sigma \bullet \sigma) \rightsquigarrow \Sigma \bullet \sigma[x[i] \mapsto v]}$$

4 Abstract domain & lattice

Domaine abstrait

Le domaine abstrait utilisé pour l'analyse des signes est défini comme suit :

$$L = \{\bot, N, Z, ZP, B, U\}$$

— $\bot$: **Bottom**, représente l'absence d'information (aucune valeur concrète possible).
— N : **Négatif**, représente les valeurs strictement négatives.
— Z : **Zéro**.
— ZP : **Strictement positif**.
— B : **Booléen**, peut être $True$ ou $False$.
— U : **Inconnu**, représente une information imprécise ou inconnue.

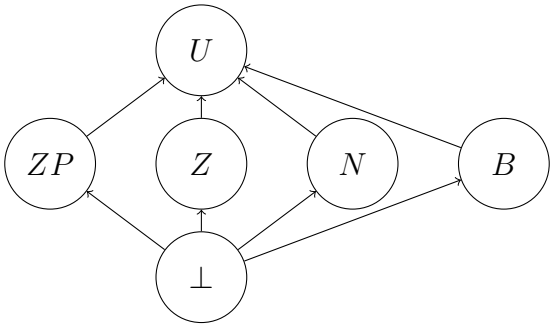
Lattice abstraite

Le domaine L est organisé en une lattice définie par un graphe de relation d'ordre partiel $(L, \sqsubseteq)$, où $\sqsubseteq$ exprime une précision croissante.

Représentation graphique

La lattice est représentée comme suit :

Relations d'ordre : $\bot \sqsubseteq N, \bot \sqsubseteq Z, \bot \sqsubseteq ZP, \bot \sqsubseteq B$ et $N, Z, ZP, B \sqsubseteq U$



Opérations sur la lattice

Les opérations principales sont définies comme suit :
— **Join ($\sqcup$) :** Le plus petit majorant commun (LUB).
— **Meet ($\sqcap$) :** Le plus grand minorant commun (GLB).
Exemples d'opérations :

$$Z \sqcup ZP = U, \quad Z \sqcap ZP = \bot,$$
$$B \sqcup ZP = U, \quad B \sqcap ZP = \bot.$$

Environnements abstraits

Un environnement abstrait est défini comme une correspondance des variables du programme à leurs valeurs abstraites :

$$\varphi : \text{Var} \to L$$

Les opérations sur les environnements abstraits incluent :
— **Join ($\sqcup$) :** Fusion des environnements, $\varphi \sqcup \varphi'$ combine les valeurs abstraites pour chaque variable.
— **Inclure ($\sqsubseteq$) :** Vérification si un environnement est inclus dans un autre ($\varphi \sqsubseteq \varphi'$).

5 Abstraction & concretisation functions

Les fonctions d'abstraction (α) et de concrétisation (γ) sont des composants essentiels de l'analyse abstraite. Dans le cadre de la **Sign Analysis**, elles relient les domaines concrets et abstraits, en s'appuyant sur une lattice contenant les éléments suivants :

$$L = \{Z, N, ZP, B, U, \bot\}$$

où :
— Z représente les valeurs strictement égales à zéro,
— N représente les valeurs strictement négatives,
— ZP représente les valeurs strictement positives,
— B représente les valeurs booléennes (`True`, `False`),
— U représente une valeur inconnue,
— $\bot$ représente le bas de la lattice, c'est-à-dire une valeur complètement indéfinie.

5.1 Fonction d'abstraction (α)

La fonction d'abstraction transforme une valeur concrète en une valeur abstraite dans le domaine L.
Définition formelle :

$$\alpha : \mathbb{Z} \cup \{\text{True}, \text{False}\} \to L$$

Règle de correspondance :

$$\alpha(v) = \begin{cases} Z & \text{si } v = 0, \\ N & \text{si } v < 0, \\ ZP & \text{si } v > 0, \\ B & \text{si } v \in \{\text{True}, \text{False}\}, \\ U & \text{sinon.} \end{cases}$$

5.2 Fonction de concrétisation (γ)

La fonction de concrétisation associe une valeur abstraite à l'ensemble des valeurs concrètes qu'elle représente.
Définition formelle :

$$\gamma : L \to \mathcal{P}(\mathbb{Z} \cup \{\text{True}, \text{False}\})$$

Règles de correspondance :

$$\begin{aligned} \gamma(Z) &= \{0\}, \\ \gamma(N) &= \{v \in \mathbb{Z} \mid v < 0\}, \\ \gamma(ZP) &= \{v \in \mathbb{Z} \mid v > 0\}, \\ \gamma(B) &= \{\text{True}, \text{False}\}, \\ \gamma(U) &= \mathbb{Z} \cup \{\text{True}, \text{False}\}, \\ \gamma(\bot) &= \emptyset. \end{aligned}$$

6 Flow functions

$$
\begin{aligned}
f[[x = y + z]](\phi) &= \phi[x \mapsto Z] && \text{if} \quad \phi(y) = Z \land \phi(z) = Z \\
&= \phi[x \mapsto ZP] && \text{if} \quad (\phi(y) = Z \land \phi(z) = ZP) \lor (\phi(y) = ZP \land \phi(z) = \\
&= \phi[x \mapsto N] && \text{if} \quad \phi(y) = N \land \phi(z) = N \\
&= \phi[x \mapsto U] && \text{otherwise}
\end{aligned}
$$

$f[[\text{array[x]} = \text{y}]](\phi) = \phi$ in all cases

7 Sound et/ou Complete ?

7.1 Is the Analysis Sound or Complete ?

Notre analyse n'est pas **sound** (correcte) elle peut être considérer comme **complete**. Voici pourquoi :

7.1.1 Pourquoi l'analyse peut être considérés intuitivement comme complète malgré ses limites apparentes

— **Approche conservatrice des valeurs abstraites** : En privilégiant l'utilisation de valeurs abstraites conservatrices (`U`), l'analyse garantit que toutes les situations d'incertitude sont couvertes par des warnings. Cette stratégie prévient l'omission d'erreurs potentielles, même dans des cas où l'information disponible est insuffisante pour déterminer une valeur exacte.

— **Gestion robuste des tableaux** : Bien que la gestion des indices repose sur des approximations abstraites, cette approche permet d'assurer la détection d'accès invalides ou non définis. Par exemple, les warnings générés pour des indices incertains ou négatifs renforcent la complétude de l'analyse en signalant des comportements suspects.

7.1.2 Pourquoi l'analyse n'est pas sound

— **Faux positifs élevés** : La présence de nombreux cas où les indices sont marqués comme `U` (inconnus) conduit à des avertissements qui ne représentent pas toujours des erreurs réelles.

— **Union des environnements** : L'utilisation d'union dans des structures conditionnelles (`if/else`) ou itératives (`while`) amplifie les faux positifs en considérant toutes les branches comme potentiellement problématiques.

7.1.3 Conclusion

Notre analyse privilégie la détection de tous les cas possibles, mais au prix de nombreux avertissements inutiles (faux positifs). Elle est donc loin d'être Sound.

7.2 Critiques et Améliorations

— **Critiques** : L'approche conservatrice génère un nombre élevé de faux positifs, ce qui peut limiter l'utilité pratique de l'analyse dans certains contextes.
— **Améliorations possibles** :
  — Une analyse plus sensible aux chemins (*path-sensitive analysis*) pourrait réduire les faux positifs en distinguant les cas où les indices sont validés dans certaines branches conditionnelles.
  — Une gestion plus précise des indices `U` notamment pour les tableau, pourrait améliorer la Soundness.

8 Type d'analyse

Notre analyse est une **Forward Analysis**. Elle s'appuie sur l'algorithme Worklist, qui traite les nœuds d'un graphe de contrôle de flux (*Control Flow Graph*, CFG) en suivant les relations entre un nœud et ses successeurs.
Justification :
— L'analyse suit le flux naturel du programme, de l'entrée principale vers les points de programme suivants, en utilisant les successeurs ($SUCC[p]$).
— Les environnements abstraits de chaque point de programme sont calculés en fonction des informations propagées depuis leurs prédécesseurs.
— L'objectif est d'atteindre un point fixe en parcourant les nœuds dans l'ordre défini par leurs connexions successives dans le CFG.

9 Types d'erreurs et d'avertissement

La Sign Analysis détecte plusieurs cas d'erreurs et d'avertissements basés sur l'évaluation des indices d'accès ou d'affectation dans des tableaux. Ces cas sont décrits ci-dessous avec les conditions sous lesquelles ils se produisent et les messages générés.

9.1 Array Assignment Cases

Lors d'une affectation à un tableau, l'indice est analysé pour déterminer s'il est valide.
— **Erreur : Indice négatif**
  — **Condition :** L'indice du tableau est évalué comme `N` (négatif).
  — **Message :** `"Array assignment with negative index detected for 'arrayName'!"`
— **Erreur : Indice booléen**
  — **Condition :** L'indice du tableau est évalué comme `B` (booléen).
  — **Message :** `"Array assignment with boolean index detected for 'arrayName'!"`
— **Avertissement : Indice potentiellement invalide**
  — **Condition :** L'indice du tableau est évalué comme `U` (inconnu).
  — **Message :** `"Array assignment with potentially invalid index for 'arrayName'."`

9.2 Array Access Cases

Lors d'un accès à un tableau, les mêmes règles que pour les affectations s'appliquent à l'analyse de l'indice.

- **Erreur : Indice négatif**
  - **Condition :** L'indice du tableau est évalué comme `N` (négatif).
  - **Message :** `"Array access with negative index detected for 'arrayName'!"`
- **Erreur : Indice booléen**
  - **Condition :** L'indice du tableau est évalué comme `B` (booléen).
  - **Message :** `"Array access with boolean index detected for 'arrayName'!"`
- **Avertissement : Indice potentiellement invalide**
  - **Condition :** L'indice du tableau est évalué comme `U` (inconnu).
  - **Message :** `"Array access with potentially invalid index for 'arrayName'."`

9.3 Notes

- Les erreurs (**Errors**) indiquent des problèmes certains, comme l'utilisation d'un indice négatif ou booléen.
- Les avertissements (**Warnings**) signalent des problèmes potentiels, comme l'utilisation d'un indice dont la valeur est inconnue.
- Ces messages sont générés par le `ResultsInterpreter` en fonction des règles d'analyse pour chaque point du programme.

10 Formalisation GEN/KILL

Cette analyse vise à déterminer, pour chaque point de programme, les indices de tableaux qui sont potentiellement en dehors des limites autorisées $[0, \text{size(array)} - 1]$. En utilisant le cadre **GEN/KILL**, l'analyse identifie et suit les variables potentiellement problématiques à travers les assignations et les accès.

10.1 GEN et KILL pour les points de programme

1. Génération (*GEN*) : Pour tout point de programme p qui effectue une assignation à une variable (indice d'un tableau), l'ensemble *GEN* capture les indices potentiellement en dehors des limites autorisées.

$$\text{GEN}[p] = \begin{cases} \{var \mid var \text{ est assigné avec une valeur } < 0 \text{ ou } \geq \text{size(array)}\} & \text{si } p \text{ est une assignation} \\ \emptyset & \text{sinon.} \end{cases}$$

2. Suppression (*KILL*) : Pour tout point de programme p, l'ensemble *KILL* capture les variables pour lesquelles l'analyse peut confirmer que leurs indices sont valides dans $[0, \text{size(array)} - 1]$.

$$\text{KILL}[p] = \begin{cases} \{var \mid var \text{ est assigné avec une valeur dans } [0, \text{size(array)} - 1]\} & \text{si } p \text{ est une assignati} \\ \emptyset & \text{sinon.} \end{cases}$$

10.2 Propagation des ensembles

Pour chaque point de programme p, les ensembles IN[p] et OUT[p] sont définis comme suit :

— IN[p] représente les variables potentiellement problématiques avant le point de programme p. Il est défini en fonction des points précédents (*predecessors*) :

$$\text{IN}[p] = \bigcup_{q \in \text{PREDECESSORS}[p]} \text{OUT}[q].$$

— OUT[p] représente les variables problématiques après le point de programme p. Il est défini à partir des ensembles *GEN* et *KILL* :

$$\text{OUT}[p] = \text{GEN}[p] \cup (\text{IN}[p] \setminus \text{KILL}[p]).$$

10.3 Analyse des structures conditionnelles et itératives

Lorsqu'un point de programme p contient une structure conditionnelle (`if/else`) ou une boucle (`while`), les environnements issus des différentes branches ou itérations sont combinés en utilisant l'**union**.

$$\text{IN}[p] = \bigcup_{q \in \text{PREDECESSORS}[p]} \text{OUT}[q]$$

Pourquoi utiliser l'union ? L'union est utilisée pour refléter une analyse de type *May* plutôt que *Must* :

— Dans une analyse de type *May*, on s'intéresse aux variables qui pourraient être problématiques dans **au moins une** des branches ou itérations. Cela garantit que toutes les possibilités d'exécution sont prises en compte, même celles qui ne se produisent pas toujours.

— Une analyse de type *Must* chercherait à identifier les variables problématiques dans **toutes** les branches, ce qui serait trop conservateur et risquerait de manquer des erreurs potentielles (C'est un choix personnel qui est le même que nous avons pris pour notre analyseur Worklist, on préfère détecter des faux positif que de rater un cas).

10.4 Détection des erreurs

Une fois les ensembles IN[p] et OUT[p] calculés pour chaque point de programme :

— Si un point de programme p est un accès à un tableau ($array[index]$) et que $index \in \text{IN}[p]$, alors une erreur est signalée :

$$\text{Erreur : Accès hors limites pour } array[index].$$

— Si un point de programme p est une assignation à un tableau ($array[index] = value$) et que $index \in \text{IN}[p]$, une erreur similaire est signalée.

10.5 Objectif de l'analyse

Cette formalisation permet :
— De suivre dynamiquement quelles variables sont potentiellement en dehors des limites autorisées.
— D'identifier les points de programme responsables des erreurs d'accès ou d'affectation.

11 Program executing procedure

Les trois programmes à fournir se trouve dans `./programmeSmall`. Pour lancer l'analyseur il suffit d'exécuter la commande :

```
java -jar  build/libs/Analyser-1.0.0-all.jar
sign-analysis programmeSmall/Program*.small
```

en remplaçant l'étoile pour analyser le programme que vous voulez analyser.

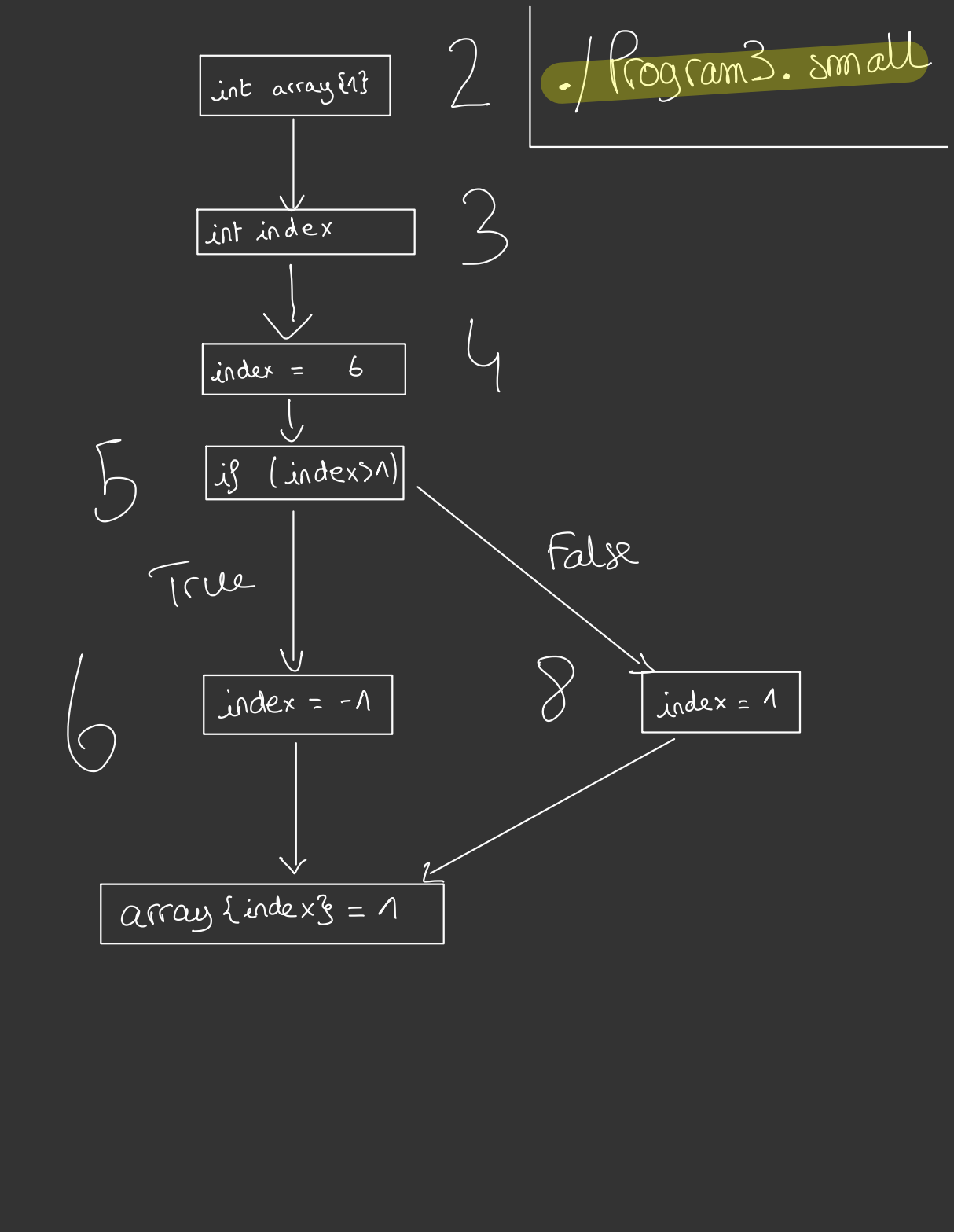
12 Extras

Nous tenons à préciser que cette analyse s'appuie sur celle de Lucas BERG, ainsi que sur l'exemple qu'il nous a fourni. Par ailleurs, nous avons rencontré une régression au niveau des scopes et, par manque de temps, nous n'avons pas pu déboguer.

13 Example de l'algorithme Worklist

Dans cette section, nous allons effectuer une démonstration pas à pas de l'analyse abstraite sur le programme suivant, nommé `Program3.small` :

```
int array{1};
int index;
index = 6;
if (index > 1) {
    index = -1;
} else {
    index = 1;
}
array{index} = 1;
```


int array[7]
2
./Program3.small
int index
3
index = 6
4
5
if (index>1)
True
False
6
index = -1
8
index = 1
array[index] = 1

PP	Work list	ϕ_p(index)	ϕ_p(array{0})	res(index)	res(array{0})
2	3	$\bot$	$\bot$	$\bot$	U
3	4	Z	$\bot$	Z	U
4	5	Z	U	ZP	U
5	6(True)/8 False	ZP	U	ZP	U
6	8/10	ZP	U	N	U
8	10	ZP	U	ZP	U
10	X	ZP	ZP	N ⊔ ZP = U	U