INF889A

Analyse de programmes pour la sécurité logicielle

Chapitre 5 - Analyse de flot d'information, analyse de dépendances

Quentin Stiévenart

Hiver 2024

Contrôle du flot d'information

On veut suivre le flot d'informations dans un programme afin de pouvoir détecter :

- des informations utilisateurs (potentiellement dangereuses) non nettoyées qui atteignent une fonction critique (`open`, `eval`, `system`, `query`)
  - problème d'intégrité
- des informations privées non nettoyées qui atteignent un point d'accès public
  - problème de confidentialité

Information dangereuse

Une information “dangereuse” :

- émane d’une **source**
- se propage dans le programme
- ne doit pas se retrouver dans un **puits** (*sink*)
  - ou seulement si elle a été *nettoyée* (*sanitized*) ou *déclassifiée*

Exemple : injection SQL

```php
$offset = $_GET['offset']; // beware, no input validation!
$query  = "SELECT id, name FROM products
                  ORDER BY name
                  LIMIT 20 OFFSET $offset;";
$result = pg_query($conn, $query);
```

(php.net)

- Source : `$_GET`
- Flot d'information : `$offset` se *propage* dans `$query`
- Puits : `pg_query`

Exemple : injection de commande

```
int main(char* argc, char** argv) {
  char cmd[CMD_MAX] = "/usr/bin/cat ";
  strcat(cmd, argv[1]);
  system(cmd);
}
```

(owasp.org)

- Source : `argv`
- Flot d'information : `argv[1]` se *propage* dans `cmd`
- Puits : `system`

Niveau de sécurité

On peut formaliser une sécurité à 2 niveaux par :

- un niveau L pour l'information sûre
- un niveau H pour l'information dangereuse
- une source retourne un niveau H
- un puits n'accepte **que** un niveau L
  - sinon, on a une violation de nos règles de sécurité

Propagation

Si on a un calcul qui dépend d'une information dangereuse, alors le résultat est dangereux

- `y = f(input())`
  - `input()` est dangereux
  - `y` l'est donc aussi
  - c'est un **flot explicite**
- de façon implicite : `y = input_bool() ? 1 : 0`
  - on a calculé la même chose que `y = input_bool()`, donc `y` doit être dangereux aussi
  - c'est un **flot implicite**
  - souvent ignoré en intégrité

Analyse de teinte

Analyse de teinte

Idée :

- une information émanant d'une source est teintée (H)
- les teintes se propagent au travers des opérations
- une opération de *sanitization* enlève la teinte
- on ne veut pas d'information teintée qui arrive dans un puits

Cela peut se faire statiquement ou dynamiquement

Identification de sources, puits, sanitization

Une difficulté en pratique est d'identifier les sources, puits, et opérations de sanitization :

- liste manuelle ?
- liste dérivée de spécifications ?
- liste *apprise* automatiquement ?

Dynamiquement : *dynamic taint tracking* code/taint.pl

On peut utiliser de l'instrumentation pour suivre dynamiquement les teintes

En Perl et Ruby (< 2.7), avec le flag `-T`

```
#!/usr/bin/perl -T
$arg = shift;
$cmd = "echo $arg";
system $cmd;
```

Résultat :

```
$ ./taint.pl foo
Insecure $ENV{PATH} while running with -T switch
  at ./taint.pl line 4.
```

Enck et al. (2014). TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. ACM TOCS.

- suit l'utilisation des données confidentielles par des applications
  - ce sont les sources : capteurs, bases de données, IMEI, ...
- but: détecter quand les données sortent du système via des *untrusted* applications
  - ce sont les puits : envois sur le réseau

Sensitive information is first identified at a taint source, where a taint marking indicating the information type is assigned. Dynamic taint analysis tracks how labeled data impacts other data in a way that might leak the original sensitive information. [...] Finally, the impacted data is identified before it leaves the system at a taint sink (usually the network interface).

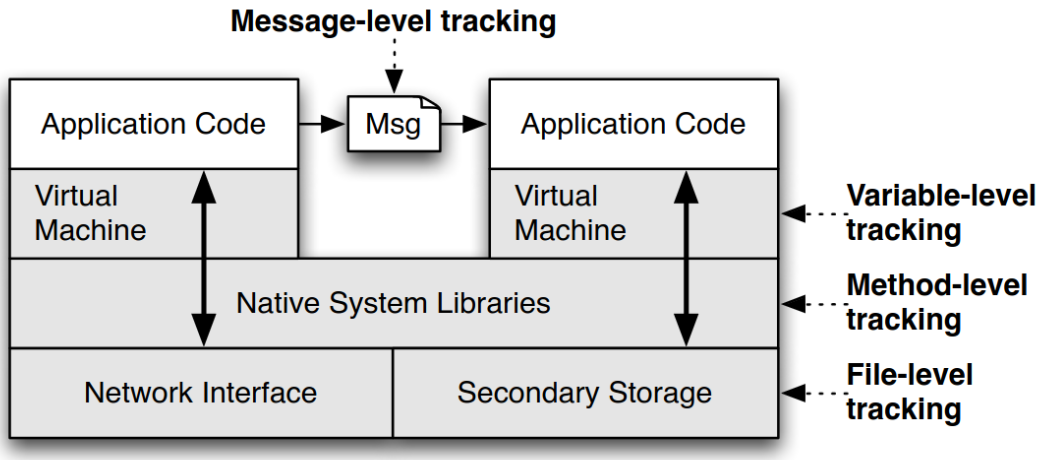
TaintDroid : approche

Enck et al., 2014

Calcule la teinte sur 4 granularités :

- *variable* : en instrumentant la VM
- *message* : entre les applications
- *méthode* : pour les bibliothèques natives via des modèles
- *fichier* : pour préserver l'information si stockage, stocké dans les attributs étendus

(Calculer la teinte au niveau des instructions a un coût trop élevé et peut perdre beaucoup de précision si le *pc* devient teinté.)



- Labels de teinte plus riches que $\{L, H\}$
  - une variable peut avoir plusieurs teintes
- Règles de propagation de teinte au niveau du bytecode

Op Format	Op Semantics	Taint Propagation	Description
const-op v_A C	$v_A \leftarrow C$	$\tau(v_A) \leftarrow \emptyset$	Clear v_A taint
move-op v_A v_B	$v_A \leftarrow v_B$	$\tau(v_A) \leftarrow \tau(v_B)$	Set v_A taint to v_B taint
move-op-R v_A	$v_A \leftarrow R$	$\tau(v_A) \leftarrow \tau(R)$	Set v_A taint to return taint
return-op v_A	$R \leftarrow v_A$	$\tau(R) \leftarrow \tau(v_A)$	Set return taint ($\emptyset$ if void)
move-op-E v_A	$v_A \leftarrow E$	$\tau(v_A) \leftarrow \tau(E)$	Set v_A taint to exception taint
throw-op v_A	$E \leftarrow v_A$	$\tau(E) \leftarrow \tau(v_A)$	Set exception taint
unary-op v_A v_B	$v_A \leftarrow \otimes v_B$	$\tau(v_A) \leftarrow \tau(v_B)$	Set v_A taint to v_B taint
binary-op v_A v_B v_C	$v_A \leftarrow v_B \otimes v_C$	$\tau(v_A) \leftarrow \tau(v_B) \cup \tau(v_C)$	Set v_A taint to v_B taint $\cup$ v_C taint
binary-op v_A v_B	$v_A \leftarrow v_A \otimes v_B$	$\tau(v_A) \leftarrow \tau(v_A) \cup \tau(v_B)$	Update v_A taint with v_B taint
binary-op v_A v_B C	$v_A \leftarrow v_B \otimes C$	$\tau(v_A) \leftarrow \tau(v_B)$	Set v_A taint to v_B taint
aput-op v_A v_B v_C	$v_B[v_C] \leftarrow v_A$	$\tau(v_B[\cdot]) \leftarrow \tau(v_B[\cdot]) \cup \tau(v_A)$	Update array v_B taint with v_A taint
aget-op v_A v_B v_C	$v_A \leftarrow v_B[v_C]$	$\tau(v_A) \leftarrow \tau(v_B[\cdot]) \cup \tau(v_C)$	Set v_A taint to array and index taint
sput-op v_A f_B	$f_B \leftarrow v_A$	$\tau(f_B) \leftarrow \tau(v_A)$	Set field f_B taint to v_A taint
sget-op v_A f_B	$v_A \leftarrow f_B$	$\tau(v_A) \leftarrow \tau(f_B)$	Set v_A taint to field f_B taint
iput-op v_A v_B f_C	$v_B(f_C) \leftarrow v_A$	$\tau(v_B(f_C)) \leftarrow \tau(v_A)$	Set field f_C taint to v_A taint
iget-op v_A v_B f_C	$v_A \leftarrow v_B(f_C)$	$\tau(v_A) \leftarrow \tau(v_B(f_C)) \cup \tau(v_B)$	Set v_A taint to field f_C and object reference taint

Pas de support des flots implicites !

À partir de 30 applications sélectionnées aléatoirement parmi le top 50 :

- utilisation manuelle de l'application avec TaintDroid activée
- enregistrement du trafic réseau pour double validation
- analyse manuelle des résultats

Résultats :

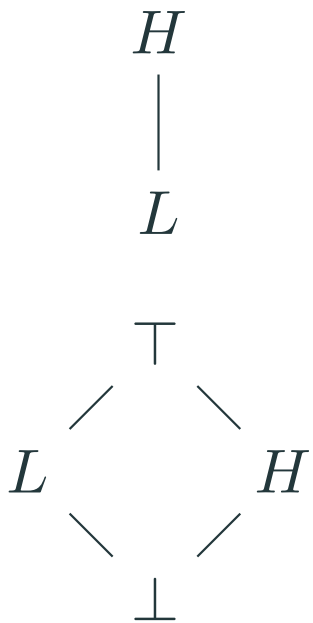
- 105 communications reportées par TaintDroid
- 2 applications envoient le numéro de téléphone, IMSI, ICC-ID, et géolocalisation
- 7 applications envoient le numéro IMEI a un serveur de contenu
- 15 applications envoient la géolocalisation à des serveurs publicitaires
- pas de faux positif observé

Performance :

- évaluée à partir de macro et micro benchmarks
- 14% plus lent

Statiquement : treillis

On peut formaliser le *niveau de sécurité* par un treillis. Par exemple, pour deux niveau de sécurité, on peut utiliser un des treillis suivants :



Treillis : différence

- Treillis 1 : $L \sqsubseteq H$
- Treillis 2 : $\bot \sqsubseteq L, H \sqsubseteq \top$

Si un puits reçoit H, quelle est la conclusion de l'analyse ?

Treillis : différence

- Treillis 1 : $L \sqsubseteq H$
- Treillis 2 : $\bot \sqsubseteq L, H \sqsubseteq \top$

Si un puits reçoit H, quelle est la conclusion de l'analyse ?

- Treillis 1 : on a *peut-être* un problème (analyse *may*)
- Treillis 2 : on a *définitivement* un problème (analyse *must*)

Statiquement : règles *dataflow*

- Si n est l'entrée :

$$[\![n]\!] = \lambda s.\bot$$

- Si n est une assignation $\mathtt{x} = a$:

$$[\![n]\!] = \lambda s.\sigma[\mathtt{x} \mapsto \mathit{eval}(\sigma, a)]$$

$$\text{où } \sigma = \textsc{join}(n)(s)$$

- Si n est une opération de sanitization `sanitize(x)` :

$$[\![n]\!] = \lambda s.\sigma[\mathtt{x} \mapsto L]$$

$$\text{où } \sigma = \textsc{join}(n)(s)$$

- Sinon :

$$[\![n]\!] = \lambda s.\textsc{join}(n)(s)$$

Avec

$$\textsc{join}(n)(s) = \bigsqcup_{p \in \mathit{pred}(n)} [\![p]\!](s)$$

Statiquement : fonction d'évaluation

$$
\begin{aligned}
\mathit{eval}(n, \sigma) &= L && \text{constante} \\
\mathit{eval}(\texttt{read()}, \sigma) &= H && \text{entrée utilisateur} \\
\mathit{eval}(\texttt{x}, \sigma) &= \sigma(x) && \text{variable} \\
\mathit{eval}(e_1 \oplus e_2, \sigma) &= \mathit{eval}(e_1, \sigma) \mathbin{\hat{\oplus}} \mathit{eval}(e_2, \sigma) && \text{opération}
\end{aligned}
$$

Où $\hat{\oplus}$ définit des règles de propagation spécifiques à chaque opération.

Exemple

```
x = read();
if (cond) {
  sanitize(x);
}
sink(x);
```

Analyse de dépendances

L'analyse de teinte pose la question suivante :

> *La valeur passée en argument à un puits dépend-elle d'une valeur provenant d'une source ?*

C'est une sous-question de l'analyse de dépendance :

> *La valeur de la variable x dépend-elle de la valeur de la variable y*

Tranchage de programme

Tranchage de programme

> *Pour un programme P et une variable x à un point p, quel est le programme minimal P' pour lequel x a la même valeur au point p ?*

C'est également une sous-question de l'analyse de dépendance :

> *La valeur de la variable x dépend-elle de la valeur de la variable y*

Beaucoup d'applications : déboguage, compréhension de programmes, ré-ingénierie, refactor, ...

Tranchage de programme, statiquement

Pour un point de programme P, on souhaite identifier tous les points de programmes P' qui influencent P

Autrement dit : quels sont les P' dont P dépend ?

P est appelé le critère de tranchage (*slicing criterion*)

Dépendances de données

```
int abs(int x) {
  int y = 0;
  if (x > 0) {
      y = x;
  }
  if (x < 0) {
      y = -x;
  }
  return y; // critère
}
```

La valeur de y dépend de x

Dépendances de données

Les dépendances de données peuvent se calculer avec des *use-definition chains* :

> *Pour un usage d'une variable y, quelles sont ses définitions ?*

- En SSA: trivial pour les "registres".
- Indirection dans la mémoire : plus complexe
  - soit on sur-approxime
  - soit on a une analyse de pointeurs

Dépendances de données

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

Quelles sont les *use-definition chains* dans ce programme ?

Dépendances de données

- `return y` dépend de `y = 0`
- `return y` dépend de `y = x`
- `return y` dépend de `y = -x`
- `y = x` dépend du paramètre `x`
- `y = -x` dépend du paramètre `x`
- `x > 0` dépend du paramètre `x`
- `x < 0` dépend du paramètre `x`

Dépendances de contrôle

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

Dépendance de contrôle = de quelle condition l'exécution de l'instruction dépend

- `y = x` dépend de `x > 0`
- `y = -x` dépend de `x < 0`

Tranchage de programme statique

Algorithme de point fixe avec liste de travail :

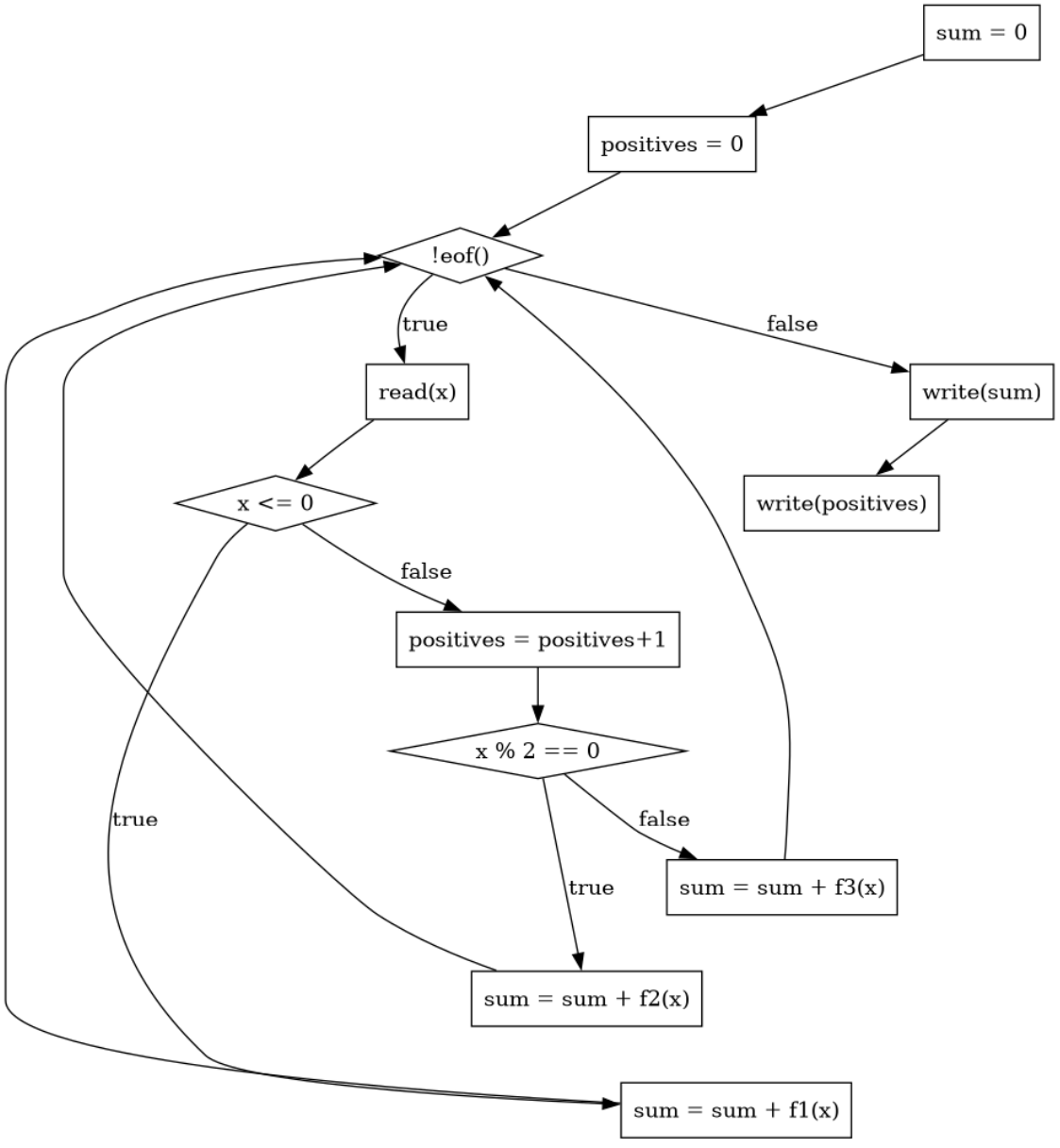
- on commence avec $W = \{p\}$ où p est notre critère de tranchage
- *slice* $= \{\}$
- tant que W n'est pas vide, prendre un élément $w \in W$:
  - ajouter w à *slice*
  - ajouter les dépendances de données de w à W
  - ajouter les dépendances de contrôle de w à W
- une fois W vide, *slice* est notre tranche

En présence de sauts inconditionnels (`goto`), ce n'est pas suffisant (cf. Agrawal, PLDI 1994)

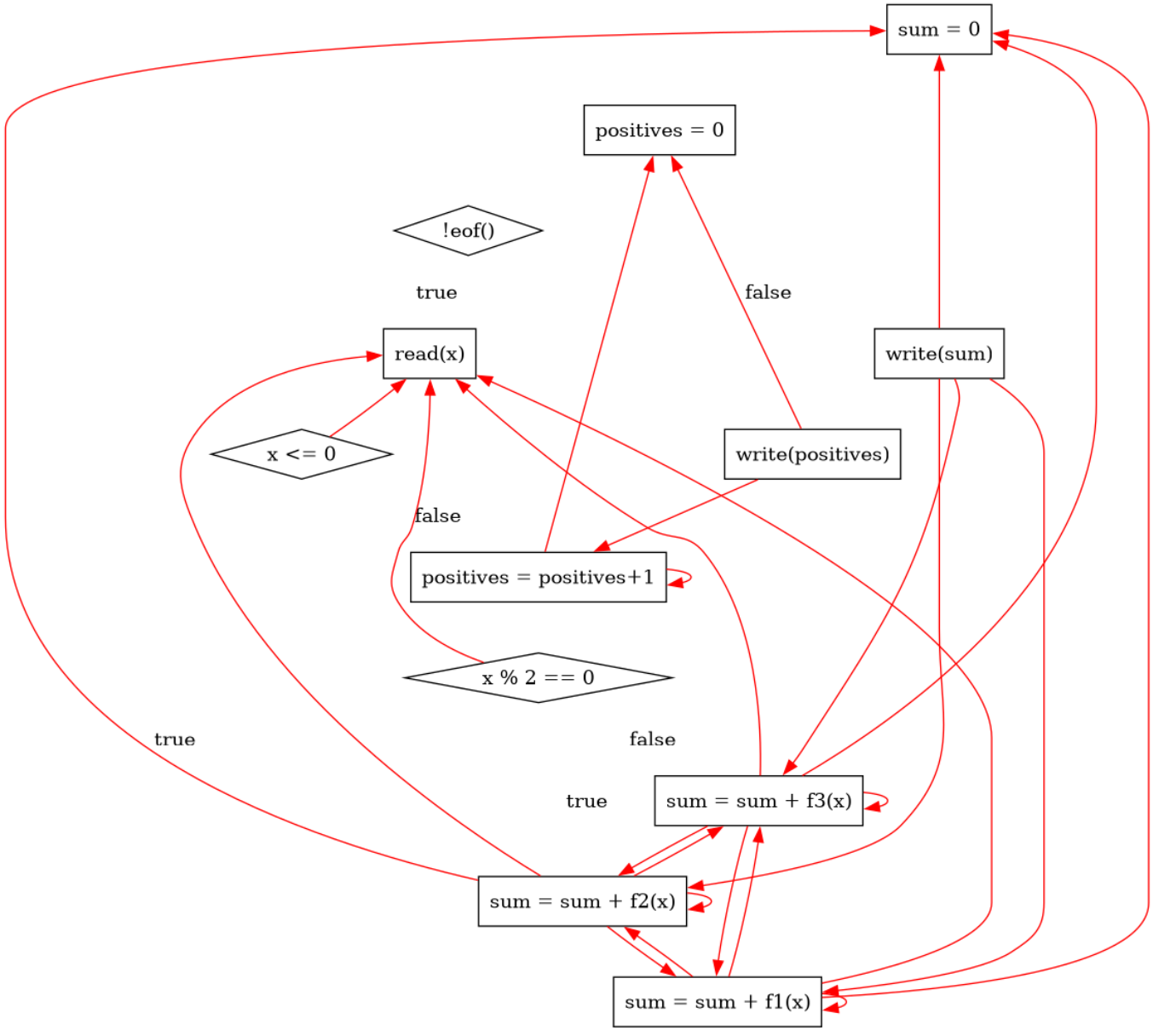
Tranchage de programme : exemple

```
sum = 0;
positives = 0;
while (!eof()) {
  read(x);
  if (x <= 0) {
    sum = sum + f1(x);
  } else {
    positives = positives + 1;
    if (x % 2 == 0) {
      sum = sum + f2(x);
    } else {
      sum = sum + f3(x);
    }
}
write(sum);
write(positives); // critère
```

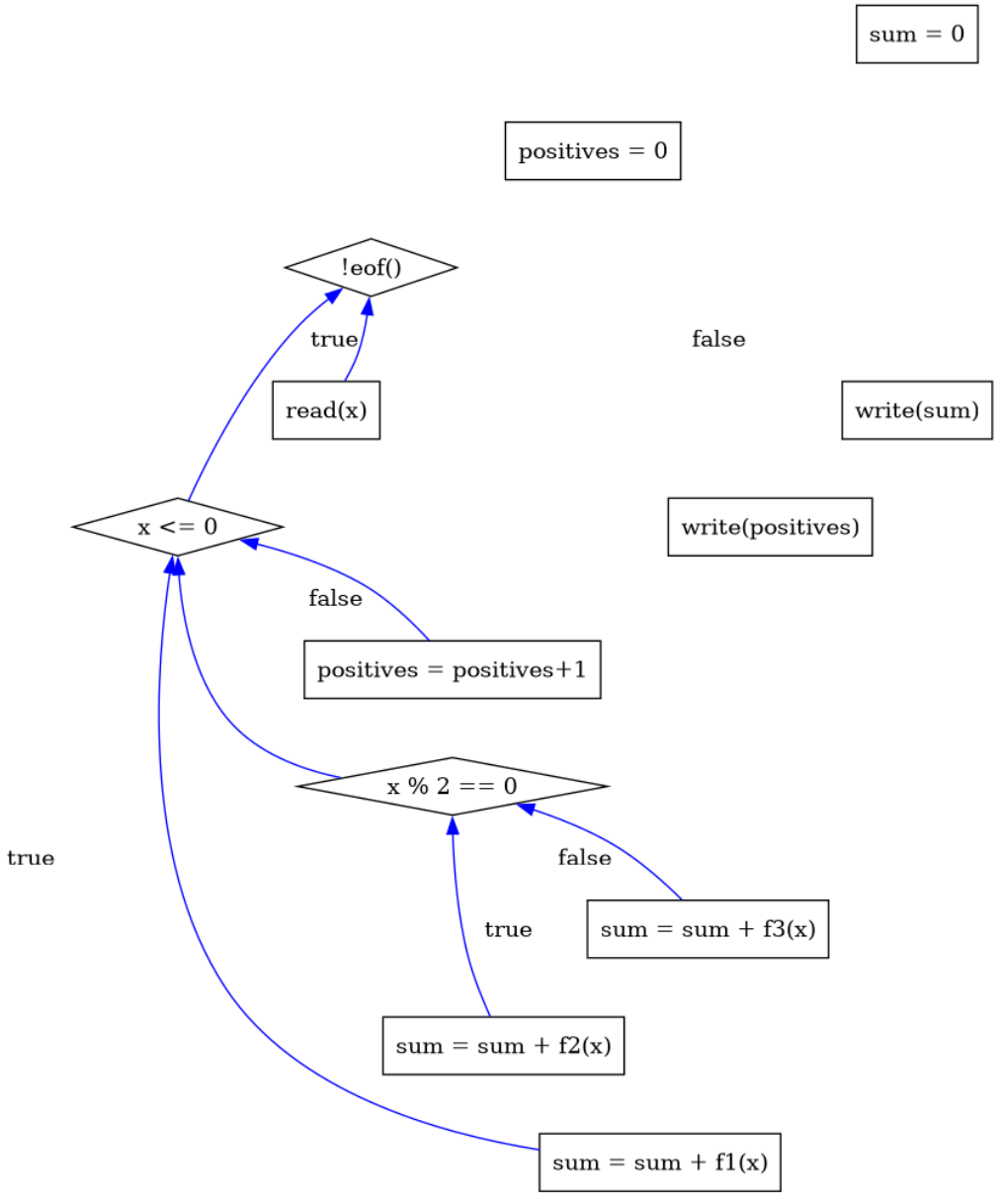
Graphe de flot de contrôle (CFG)



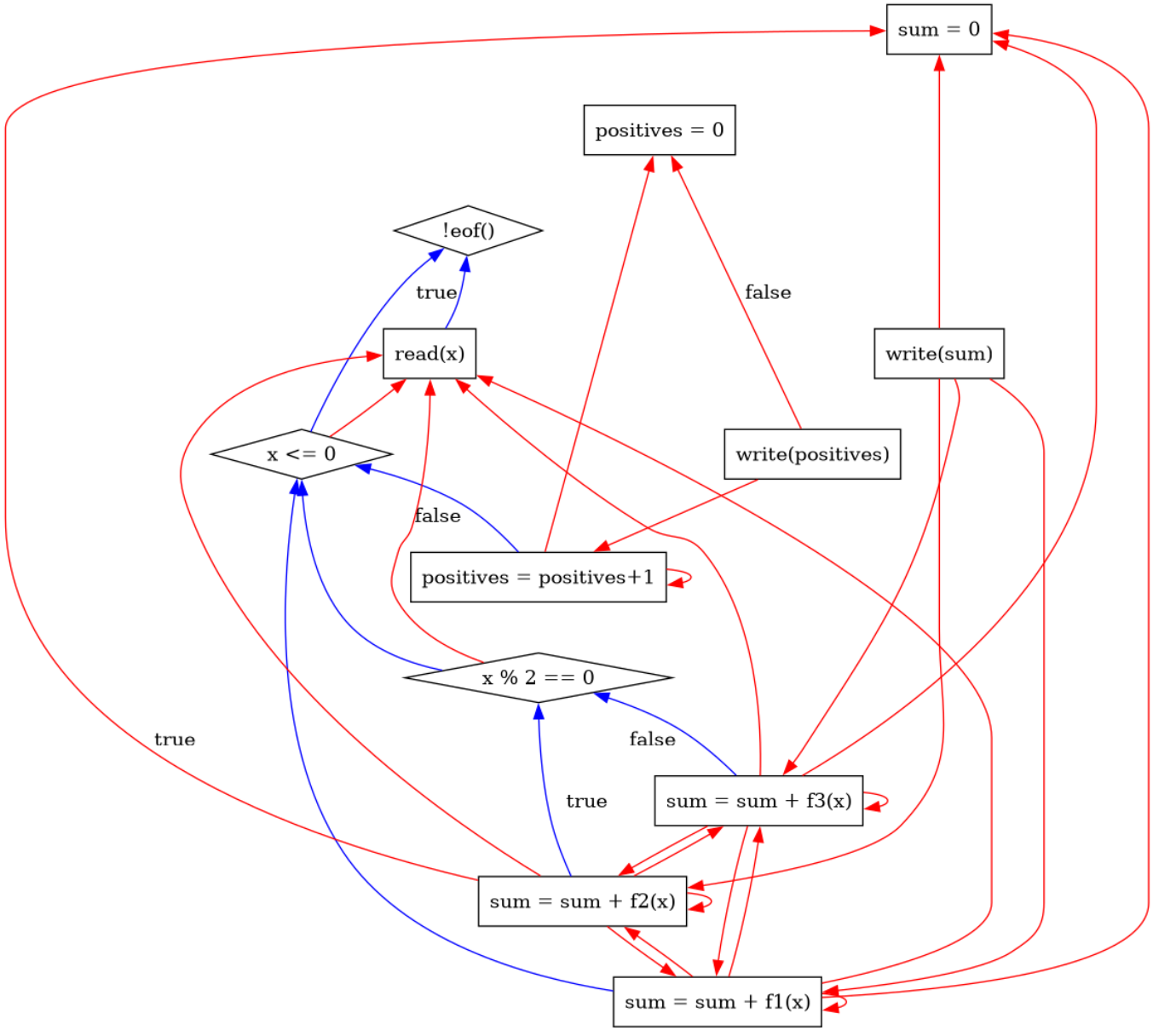
Graphe de dépendances de données (DDG)



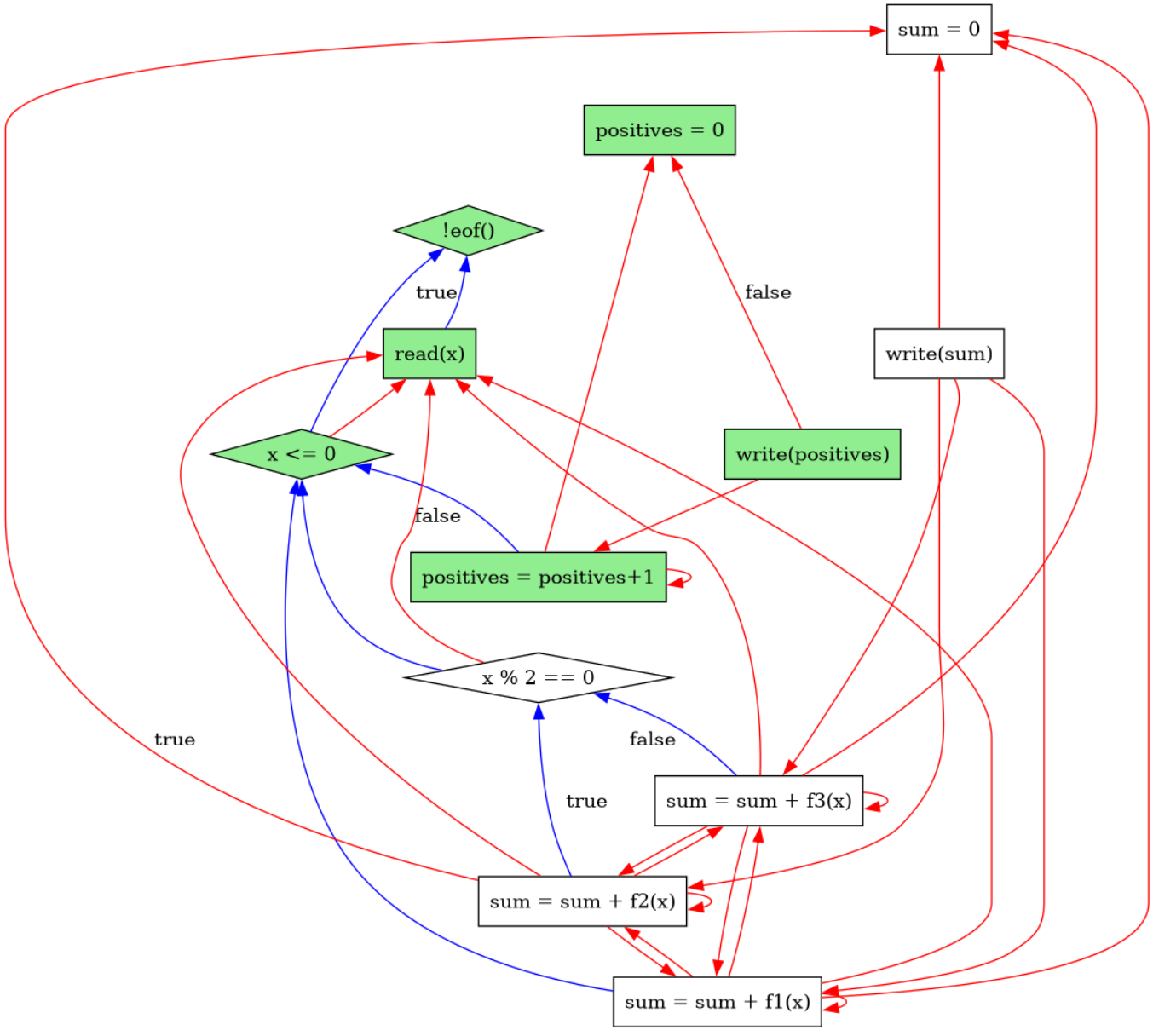
Graphe de dépendances de contrôle (CDG)



Graphe de dépendances de programme (PDG)



Tranche de programme



Tranche de programme

Tranche :

```
positives = 0;
while (!eof()) {
  read(x);
  if (x <= 0) {
  } else {
    positives = positives + 1;
  }
}
write(positives); // critère
```

Exercice

Exercice : calculer la tranche avec comme critère `write(sum)`

Tranchage de programme : éléments avancés

On pourrait vouloir cette tranche à la place :

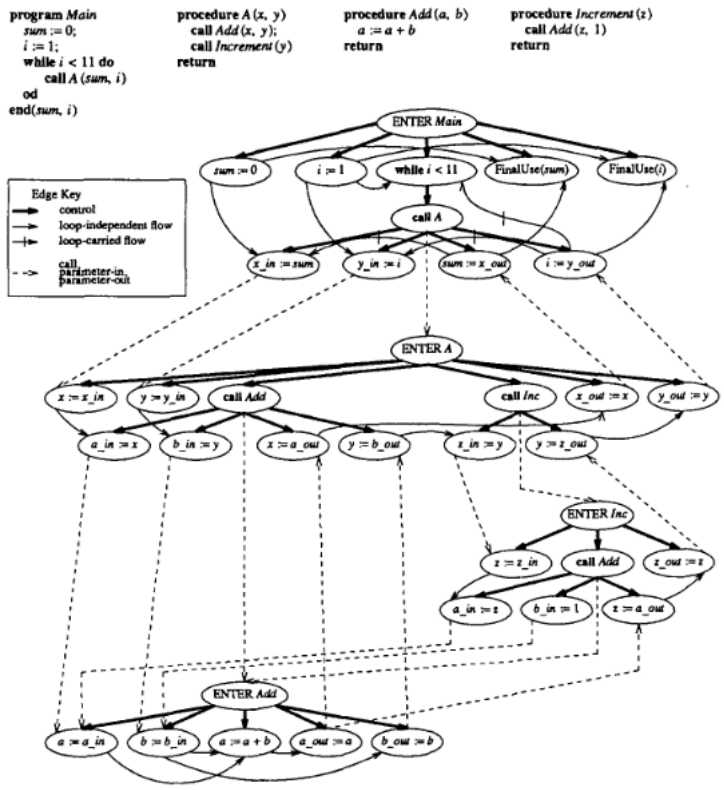
```
positives = 0;
while (!eof()) {
  read(x);
  if (x > 0) {
    positives = positives + 1;
  }
}
write(positives); // critère
```

Mais le programme a donc été modifié. C'est du **amorphous slicing**

Harman, M., & Danicic, S. (1997). Amorphous Program Slicing. IWPC.

Tranchage de programme : éléments avancés

On peut généraliser le graphe de dépendances de programme pour du slicing inter-procédural



Horwitz, S., Reps, T., & Binkley, D. (1990). Interprocedural Slicing using Dependence Graphs. TOPLAS.

Tranchage de programme : éléments avancés

On peut faire cela dynamiquement :

- slicing dynamique : même méthode que le statique, mais on a une vue partielle des dépendances pour les exécutions effectuées

  Agrawal, H., & Horgan, J. R. (1990). Dynamic program slicing. ACM SIGPLAN Notices.
- slicing basé sur les observations : on exécute un fragment et on regarde si le résultat est différent

  Binkley, D., Gold, N., Harman, M., Krinke, J., & Yoo, S. (2013). Observation-Based Slicing. RN.

Articles

Analyse de teinte :

- Grech, & Smaragdakis. (2017). P/Taint: Unified Points-To and Taint Analysis. OOPSLA.
- Pauck, Bodden, & Wehrheim. (2018). Do Android Taint Analysis Tools Keep Their Promises?. ESEC/FSE.
- Karim, Tip, Sochůrková, & Sen. (2018). Platform-Independent Dynamic Taint Analysis for JavaScript. IEEE Transactions on Software Engineering.
- Muralee, et al. (2023). ARGUS: A Framework for Staged Static Taint Analysis of GitHub Workflows and Actions. Usenix Security.
- Luo, Li, & Meng. (2022). TChecker: Precise Static Inter-Procedural Analysis for Detecting Taint-Style Vulnerabilities in PHP Applications. CCS.

Articles

Slicing :

- Salimi, S., & Kharrazi, M. (2022). VulSlicer: Vulnerability detection through code slicing. JSS.
- Binkley, D. et al. (2019). A Comparison of Tree- and Line-Oriented Observational Slicing. ESE.
- Ye, M. et al. (2018). TZSlicer: Security-Aware Dynamic Program Slicing for Hardware Isolation. HOST.
- Stiévenart, Q. et al. (2022). Static Stack-Preserving Intra-Procedural Slicing of WebAssembly Binaries. ICSE.