INF889A

Analyse de programmes pour la sécurité logicielle

Extra - Graphes de propriétés de code et Joern

Quentin Stiévenart

Hiver 2024

Code Property Graphs

Yamaguchi, F., et al. (2014). Modeling and Discovering Vulnerabilities with Code Property Graphs. IEE S&P.

> *We introduce a novel representation of source code called a code property graph that merges concepts of classic program analysis, namely abstract syntax trees, control flow graphs and program dependence graphs. This comprehensive representation enables us te elegantly model templates for common vulnerabilities with graph traversals that, for instance can identify buffer overflows, integer overflows, format string vulnerabilities, or memory disclosures.*

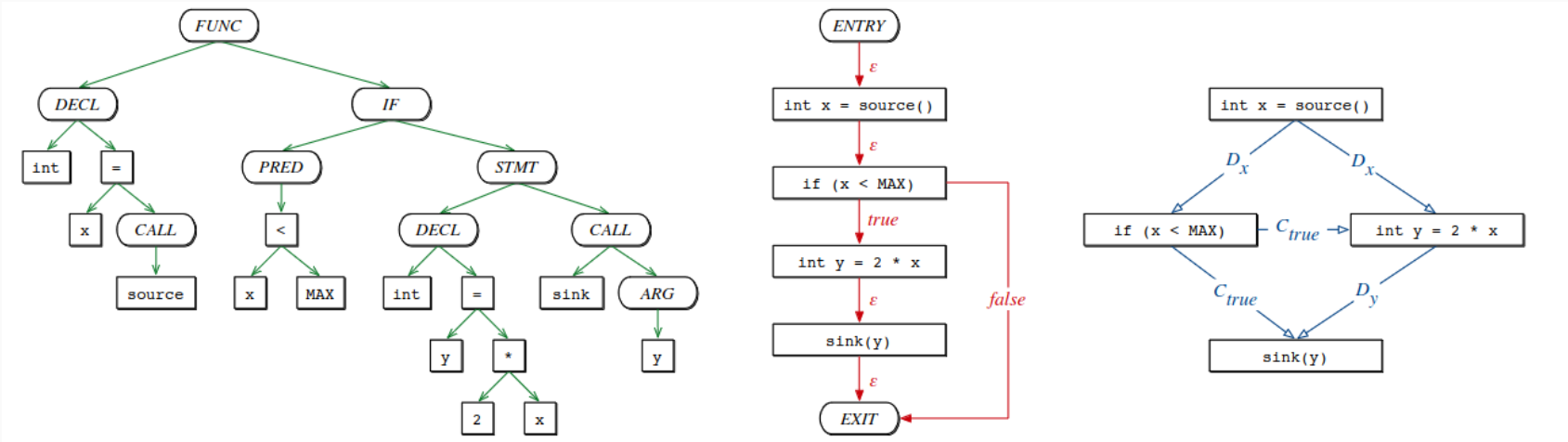
Code Property Graphs

- But : analyser beaucoup de code
- Idée : beaucoup de vulnérabilités peuvent être découverte en combinant la structure, le control flow et les dépendances ans le code.
- Approche : regrouper plusieurs représentations sous forme d'un seul graphe, faire des requêtes sur ce graphe (*graph traversals*)
- Applicatio :
  - Analyse du noyau Linux. La majorité des vulnérabilités trouvées en 2012 peuvent être décrites comme traversées de graphe
  - Utilisation d'une base de donnée graphes pour être efficace
  - 18 vulnérabilités identifiées dans le noyau Linux

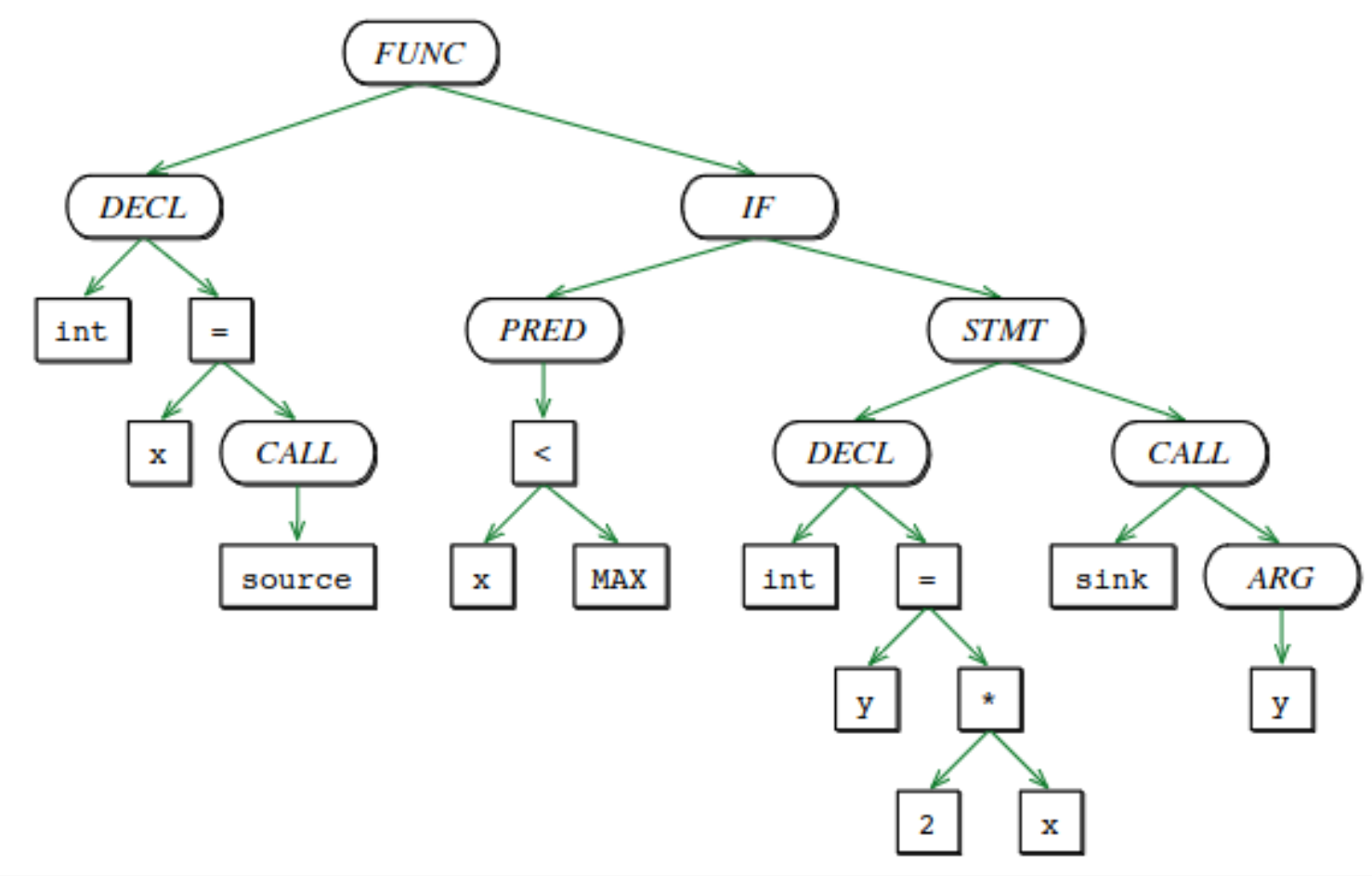
Exemple

```
void foo() {
  int x = source();
  if (x < MAX) {
    int y = 2 * x;
    sink(y);
  }
}
```

Représentations de programme



Abstract Syntax Tree (AST)



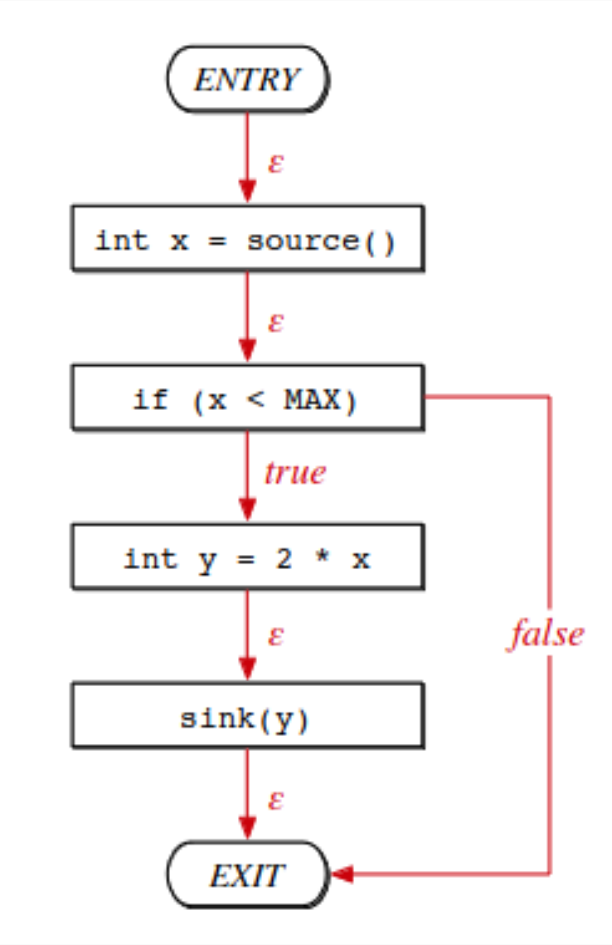
Abstract Syntax Tree (AST)

Représentation abstraite du programme après l'avoir *parsé* :

- nœuds internes : opérateurs
- feuilles : opérandes

Pas adapté pour des analyses avancées

Control-Flow Graph (CFG)



Control-Flow Graph (CFG)

Décrit l'ordre dans lequel les opérations sont exécutées

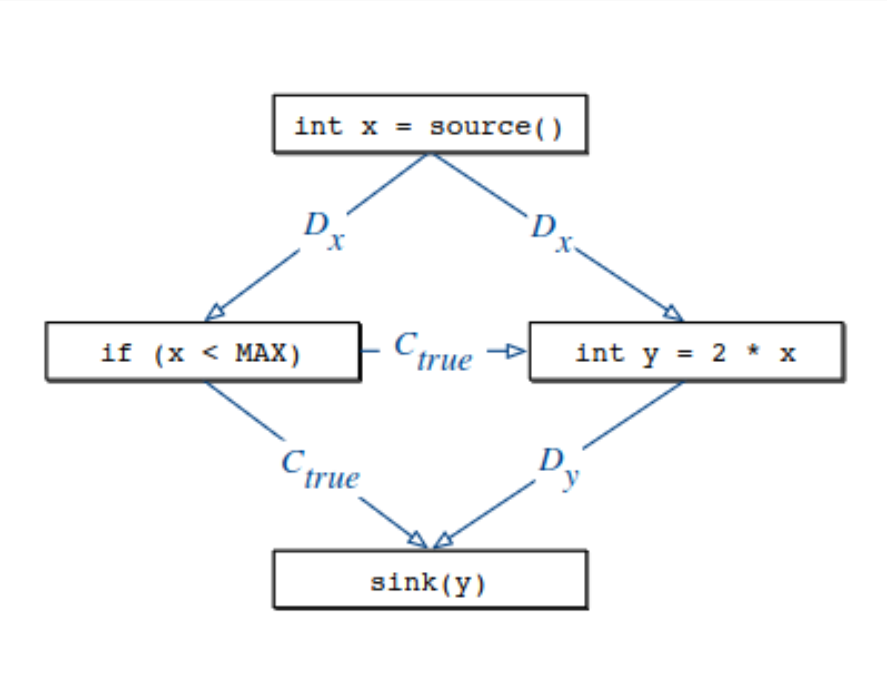
- Nœuds : instructions et prédicats
- Arcs : transfert de contrôle (non ordonnées, mais annotées avec T, F, ou ϵ)

Construit à partir de l'AST.

Ne donne pas d'informations sur les dépendances de données

- on en a besoin pour identifier les instructions que contrôle un attaquant

Program Dependence Graph (PDG)



Program Dependence Graph (PDG)

Représente explicitement les dépendances entres instructions et prédicats

Deux types d'arcs :

- dépendances de données
- dépendances de contrôle

Construit à partir du CFG avec une analyse de définition-usage (*def-use*) et une analyse de définition atteignante (*reaching definitions*)

Graphes de propriétés

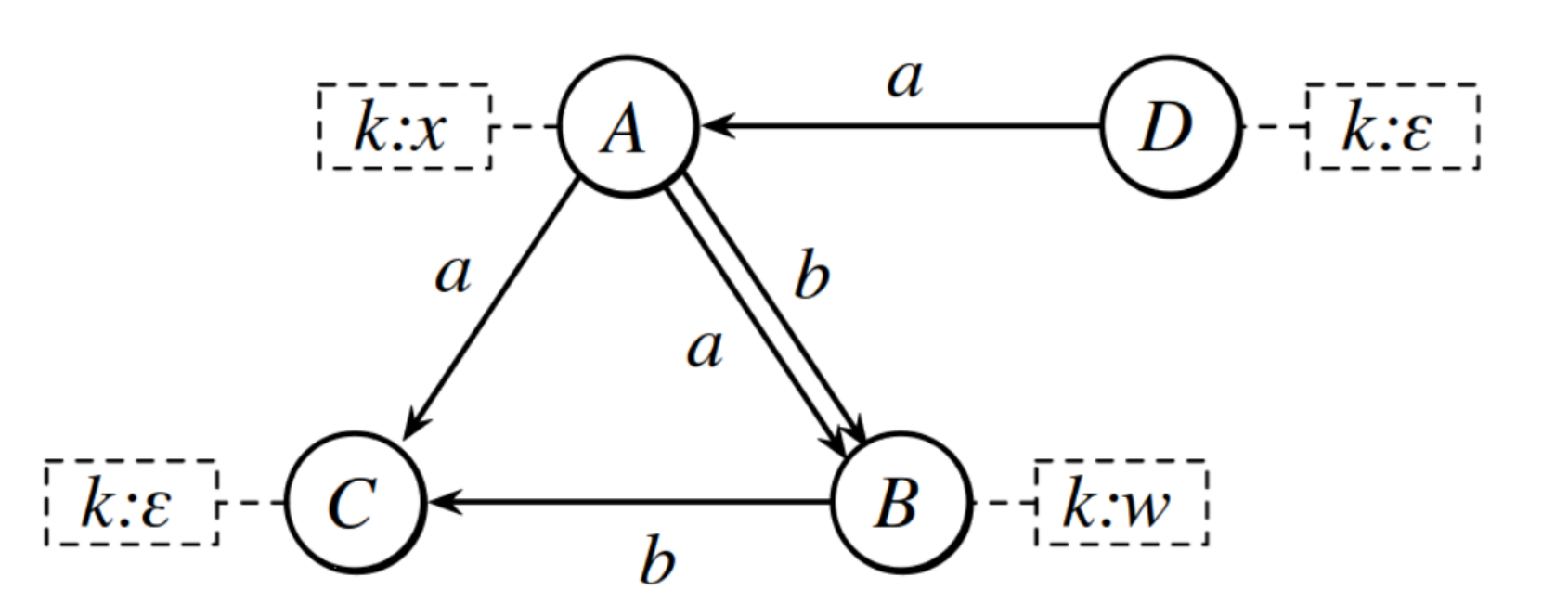
Concept introduit par Rodriguez & Neubauer (2011), utilisé dans de nombreuses bases de données graphes (Neo4J, ...)

Définition : *graphe de propriété*

Un graphe de propriété $G = (V, E, \lambda, \mu)$ possède :

- des nœuds V
- des arcs E
- une fonction d'étiquetage d'arcs : $\lambda : E \to \Sigma$
- une fonction $\mu : (V \cup E) \times K \to S$ qui assigne des propriétés
  - K sont les clés des propriétés
  - S sont les valeurs des propriétés

Graphes de propriétés : exemple



Observations :

- c'est un *multi-graphe* : on peut avoir plusieurs arcs entre deux nœuds
- l'interprétation des nœuds, arcs, et propriétés et laissée libre à l'utilisation

Traversée de graphes de propriétés

Définition : *traversée*

Une traversée est une fonction $\mathcal{T} : \mathcal{P}(V) \to \mathcal{P}(V)$ qui associe un ensemble de nœuds à un autre ensemble de nœuds

Définition très souple.

Les traversées se composent : $\mathcal{T}_0 \circ \mathcal{T}_1$

Exemples de traversées

Garder les nœuds qui satisfont à un prédicat p :

$$\textsc{Filter}_p(X) = \{v \in X : p(v)\}$$

Avancer dans le graphe :

$$\textsc{Out}(X) = \bigcup_{v \in X} \{u : (v, u) \in E\}$$

Reculer dans le graphe :

$$\textsc{In}(X) = \bigcup_{u \in X} \{v : (v, u) \in E\}$$

Combiner des traversées

$$\textsc{Or}(\mathcal{T}_1, \dots, \mathcal{T}_n)(X) = \mathcal{T}_1(X) \cup \dots \cup \mathcal{T}_n(X)$$
$$\textsc{And}(\mathcal{T}_1, \dots, \mathcal{T}_n)(X) = \mathcal{T}_1(X) \cap \dots \cap \mathcal{T}_n(X)$$

Graphes de propriétés en pratique

Gremlin : langage de requête utilisé par Neo4j, OrientDB, Hadoop, ...

```
g.V().has("name","gremlin").as("a").
  out("created").in("created").
  where(neq("a")).
  in("manages").
  groupCount().by("name")
```

Représentation de l'AST en graphe de propriétés

$$G_A = (V_A, E_A, \lambda_A, \mu_A)$$

- V_A : les nœuds de l'AST
- E_A : arcs représentant les connexions de l'AST
- λ_A : assigne un label "AST" à chaque arc
- μ_A : assigne des propriétés au nœuds
  - $K = \texttt{code}, \texttt{order}$
  - `code` est l'opérateur/opérande représentée par le nœud
  - `order` indique l'ordre des enfants d'un nœud

Représentation du CFG en graphe de propriétés

$$G_C = (V_C, E_C, \lambda_C, \cdot)$$

- $V_C \subseteq V_A$: les nœuds de l'AST ayant la propriété `code` mise à `STMT` ou `PRED`
- E_C : les arcs du CFG
- λ_C assigne les labels $\{\texttt{true}, \texttt{false}, \epsilon\}$

Représentation du PDG en graphe de propriétés

$$G_P = (V_C, E_P, \lambda_P, \mu_P)$$

- E_P : les arcs de dépendances
- λ_P assigne les labels $\{C, D\}$
- μ_P assigne des propriétés aux arcs :
  - `symbol` : dépendance de donnée via un symbole
  - `condition` : `true` or `false` selon la valeur de la condition

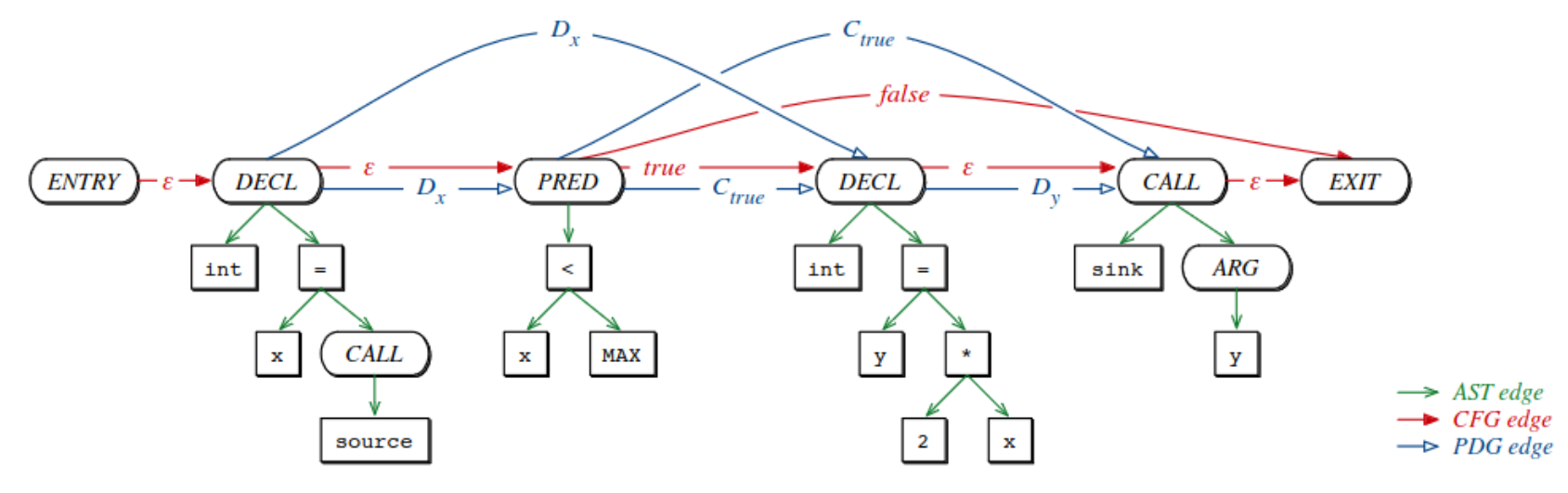
Le tout mis ensemble

Définition : *code property graph*

Un CPG est un graphe de propriété $G = (V, E, \lambda, \mu)$ où :

- $V = V_A$,
- $E = E_A \cup E_C \cup E_P$,
- $\lambda = \lambda_U \cup \lambda C \cup \lambda P$,
- $\mu = \mu_A \cup \mu_P$

Exemple de CPG



Utilisation du CPG pour la détection de vulnérabilités

Vieille implémentation de SSH (iOS) :

```
if (channelp) {
  uint32_t namelen =
    _libssh2_ntohu32(data + 9 + sizeof("exit-signal"));
  channelp->exit_signal = LIBSSH2_ALLOC(session, namelen+1);
  ...
  memcpy(channel->exit_signal,
         data + 13 + sizeof("exit_signal"), namelen);
  channelp->exit_signal[namelen] = '\0';
  ...
}
```

- `data` est contrôlé par l'utilisateur
- donc `namelen` aussi
- si `namelen` est choisi à $2^{32} - 1$, on alloue 0 octets
- on a une buffer overflow au `memcpy`

Détection de la vulnérabilité

On va procéder en trois étapes :

- détection syntaxique
- détection avec le flot de contrôle
- détection avec les dépendances

Détection syntaxique

Pour détecter la vulnérabilité syntaxiquement :

1. On cherche un appel à une fonction d'allocation
2. Qui contient une somme

Détection de nœuds

$$\textsc{TNodes}(X) = \bigcup_{v \in X} \left(v \cup \left(\bigcup_{v_c \in \textsc{Out}_{\texttt{AST}}(\{v\})} \textsc{TNodes}(\{v_c\}) \right) \right)$$

$$\textsc{Match}_p(V) = \textsc{Filter}_p \circ \textsc{TNodes}(V)$$

- $\textsc{TNodes}$: traverse tous les nœuds de l'AST
- $\textsc{Match}_p$: traverse tous les nœuds de l'AST qui satisfont à un prédicat
  - par exemple, "un appel à une fonction d'allocation"

Arguments

Pour trouver le $i$ème argument qui satisfait à un prédicat p , on fera :

$$\textsc{Match}_p \circ \textsc{Arg}_f^i$$

Détection syntaxique

Avec cela, on peut déjà faire pas mal :

- Arguments dangereux, e.g., format strings, avec $p(v)$ vrai si v n'est pas une chaîne constante :

$$\textsc{Match}_p \circ \textsc{Arg}^1_{\texttt{sprintf}}$$

- Débordement arithmétique, en particulier dans les allocations, avec p vrai si v est une opération arithmétique

$$\textsc{Match}_p \circ \textsc{Arg}^1_{\texttt{malloc}}$$

- Problème de typage d'entiers : un peu plus compliqué (voir papier)

Détection syntaxique

Les descriptions de vulnérabilité syntaxiques sont donc exprimés sous forme de Match

Mais c'est insuffisant pour exprimer certains éléments :

- "la valeur doit être contrôlée par l'attaquant"
- l'utilisation de variables intermédiaires

Détection avec le CFG

En prenant le CFG on compte, on peut parler de chemin dans le CFG.

Cela permet d'exprimer par exemple :

- une fuite de ressource : une allocation qui n'est pas libérée sur tous les chemins par exemple
- un lock non relaché sur tous les chemins
- problèmes de type *use-after-free* : une utilisation sur un chemin où une ressource a été libérée

Elles peuvent s'exprimer via une traversée DFS du graphe :

- on part d'un nœud source (décrit syntaxiquement)
- on arrive à un nœud destination (décrit syntaxiquement)
- on ne passe pas par un certain type de nœud (décrit syntaxiquement)

Détection avec le PDG

Pour trouver des vulnérabilités de type *taint*, on veut également suivre le flot de données

- $\textsc{Statement}$ trouve les statements qui contiennent un nœud
- $\textsc{Producers}_N$ trouve les variables utilisées dans un statement
- $\textsc{Sources}$ trouve les statements qui produisent de la donnée utilisée dans un statement

Exemple

On veut détecter le résultat de `get_user` (contrôlé par l'attaquant) qui arrive en 3ème argument à `memcpy`

$$\textsc{Match}_p \circ \textsc{SourcesArg}^3_{\textsc{memcpy}}$$

où p est vrai pour les appels à `get_user`

Sanitization

Problème : on va avoir des faux positifs pour les cas où la donnée a été validée

Solution : introduction de *sanitizers*, et définition de UNSANITIZED, qui est comme SOURCES, mais seulement si ce n'est pas sanitized.

Description de vulnérabilité de style *taint*

On a donc trois éléments :

- une description des sources
- une description des puits
- une description des sanitizers

Vulnérabilités de style *taint*

- Buffer overflow, où $\mathcal{T}_s$ représente les sanitizers

$$\textsc{Arg}^{1}_{\texttt{get\_user}} \circ \textsc{Unsanitized}_{\{\mathcal{T}_s\}} \circ \textsc{Arg}^{3}_{\textsc{memcpy}}$$

- Injection de code

$$\textsc{Arg}^{2}_{\texttt{recv}} \circ \textsc{Unsanitized}_{\{\mathcal{T}_s\}} \circ \textsc{Arg}^{1}_{\textsc{system}}$$

Évaluation

Un outil a été implémenté :

- analyse de C(++)
- calcule l'AST, CFG, et PDG
- liaison des nœuds basé sur la structure d'appel visible (pointeurs de fonctions ignorés)

Application au noyau Linux

- 1.3 millions de LOC
- 52 millions de nœuds
- 87 millions d'arcs
- 110 minutes sur un laptop avec Neo4j
- 14GB d'espace disque

Vulnérabilités étudiées

Catégorisation des CVE du noyau Linux en 2012 :

- 88 vulnérabilités
- inspection manuelle pour trouver la cause
- résumé en 12 types de vulnérabilités

Vulnerability types	Code representations			
	AST	AST+PDG	AST+CFG	AST+CFG+PDG
Memory Disclosure				✓
Buffer Overflow		(✓)		✓
Resource Leaks			✓	✓
Design Errors				
Null Pointer Dereference				✓
Missing Permission Checks		✓		✓
Race Conditions				
Integer Overflows				✓
Division by Zero		✓		✓
Use After Free			(✓)	(✓)
Integer Type Issues				✓
Insecure Arguments	✓	✓	✓	✓

Traversées implémentées

Implémentation de traversées pour :

- *memory disclosures* (mauvaise initialisation de structure copiée en *user space*) : le type le plus important en 2012
  - traversée de type taint
- *buffer overflow* : second type le plus important en 2012
  - traversée de type taint
- *memory mapping* : problème spécifique aux noyaux
  - traversée syntaxique
- *zero-byte allocation*
  - traversée de type taint

18 nouvelles vulnérabilités trouvées (10 CVE)

Détection de buffer overflow

$$\mathcal{T}_0 \circ \textsc{Unsanitized}_{\mathcal{T}_1^s} \circ \mathcal{T}_2$$

- $\mathcal{T}_0$ représente les sources contrôlées par l'attaquant
  - $\mathcal{T}_0^0 = \textsc{Arg}^1_{\texttt{copy\_from\_user}}$ car copie une structure utilisateur dans l'espace noyau
  - $\mathcal{T}_0^1 = \textsc{Func}(\texttt{\_write}) \circ \textsc{Param}_{\texttt{cnt}}$, les arguments des *syscalls* d'écriture
- $\mathcal{T}_1^s$ représente les sanitizers
  - émet l'hypothèse qu'il n'y a pas de problème si le buffer de destination est dynamiquement alloué, ou si il y a une expression relationnelle pour calculer la taille du buffer.
- $\mathcal{T}_2$ représente les puits
  - $\mathcal{T}_2^0 = \textsc{Arg}^3_{\texttt{memcpy}}$
  - $\mathcal{T}_2^1 = \textsc{Arg}^3_{\texttt{copy\_from\_user}}$

Détection de buffer overflow

$$\textsc{Or}(\mathcal{T}_0^0 \circ \textsc{Unsanitized}_{\mathcal{T}_1^s} \circ \textsc{Or}(\mathcal{T}_2^0, \mathcal{T}_2^1),$$
$$\mathcal{T}_0^1 \circ \textsc{Unsanitized}_{\mathcal{T}_1^s} \circ \mathcal{T}_2^1)$$

Limitations

- Certaines vulnérabilités restent impossible à modéliser (e.g., conditions de course)
- Pas d'analyse inter-procédurale (travaux futurs)
- Graphe d'appel limité
- Pas d'information sur le temps de calcul pour les traversées
- Dommage de ne pas modéliser plus de traversées liées aux vulnérabilités les plus importantes

Outil: Joern

Importer le code

```
> importCode("./src")
> cpg
```

Montrer les graphes

```
> cpg.method.name("main").dotAst
> cpg.method.name("main").dotCfg
> cpg.method.name("main").dotCpg14
```

Montrer les graphes

```
$ joern-export --repr=all --format=dot
$ joern-export --repr=all --format=neo4jcsv
$ joern-export --repr=all --format=graphml
```

Naviguer dans le graphe

```
> cpg.metaData
```

Applique `.metaData` à chaque nœud du graphe (pour ceux qui en ont un)

Évalué de façon paresseuse

```
> cpg.metaData.toList # ou .l pour plus court
```

Détecter le langage utilisé

```
> cpg.metaData.language.l
```

Lister les fonctions

```
> cpg.method.l
> cpg.method.name.l
```

Filtrer les résultats

```
> cpg.method.filter(_.isExternal == false).name.l
> cpg.method.isExternal(false).name.l
```

Chercher des expressions régulières

```
> cpg.call.name("str.*cpy").l
> cpg.call.name("f?exec(v|l)p?e?(at)?").l
> cpg.call.code(".*argc.*strcmp.*").l
```

Itérer sur les résultats

```
> cpg.method.map(n => (n.isExternal, n.name)).l
```

Naviguer dans l'AST

```
> cpg.call.name("printf").astChildren.isLiteral.code.l
```

Répéter des opérations

```
> cpg.method.name("main")
    .repeat(_.astChildren)(_.maxDepth(5)).l
> cpg.method.name("main")
    .repeat(_.astChildren)(_.until(_.isLiteral)).l
```

Traverser l'AST

```
> cpg.method.name("main").ast.l
> cpg.method.name("main").ast.isCall.code.l
```

Traverser l'AST

Il existe plein d'opérateurs pour traverser/filtrer l'AST

```
> cpg.method("main").ast.isCall.code.l
> cpg.method("main").ast.isControlStructure.code.l
```

Avec des décorateurs qui généralisent les concepts à travers plusieurs langages :

```
> cpg.call.name("source").inAstMinusLeaf
      .isCall.name(".*assignment.*").argument(1).l
> cpg.call("source").inAssignment.target.l
```

Exemple : détecter les `if` avec une condition constante

```
> cpg.ifBlock.condition.isLiteral.astParent.l
```

Utiliser de graphe d'appel

Appels sortants :

```
> cpg.method.name("main").callee
```

Appels entrants :

```
> cpg.method.name("printf").caller
```

Utiliser le CFG

Pour savoir quel autre nœud un nœud contrôle et inversement

```
> cpg.call.code(".*argv.*").controls.l
> cpg.call.name("strcpy").controlledBy.length
```

On a aussi accès à `dominate/dominatedBy`, `postDominate/postDominatedBy`

Requêtes de type data flow

```
> def source = cpg.method.name("main").parameter
> def sink = cpg.call.name("strcpy").argument
> sink.reachableBy(source).l
```

Pour avoir le chemin :

```
> sink.reachableByFlows(source).p
```

Implémentation de `reachableBy`

```
/** Recursively expand the DDG backwards and return a list
  * of all results, given by at least a source node in
  * `sourceSymbols` and the path between the source
  * symbol * and the sink.
  *
  * This method stays within the method (intra-procedural
  * analysis) and terminates at method parameters and at
  * output arguments.
  */
```

joern/dataflowengineoss/queryengine/TaskSolver.scala

Autres possibilités

- ajouter des données à des nœuds (`newTagNode`)
- définir de nouvelles traversées
- ajouter de la sémantique
- obtenir de l'aide : `cpg.method("main").help`

En pratique

Une base de données de requêtes : https://queries.joern.io/ (pas très fournie)

`joern-scan` qui automatise les requêtes

Exemple: constant-array-access-no-check pour détecter des *heap buffer overflow*

Array access at fixed offset but sufficient length check not determined

scanners/c/MissingLengthCheck.scala

Exemple

```
cpg.method.arrayAccess.filter { access =>
  val arrName = access.simpleName
  arrName.nonEmpty && !arrName.forall(x =>
    access.method.local.nameExact(x).nonEmpty)
}.usesConstantOffset.flatMap { arrayAccess =>
  val lenFields =
    potentialLengthFields(arrayAccess, arrayAccess.method)
  if (lenFields.nonEmpty) {
    List((arrayAccess, lenFields))
  } else {
    List()
  }
}.collect {
  case (arrayAccess, lenFields) if !checked(arrayAccess, lenFields) =>
    arrayAccess
}
```

En pratique

elManto/StaticAnalysisQueries

> *A set of queries to find CVEs*

Exemple : CVE-2019-1010315

> *WavPack 5.1 and earlier is affected by: CWE 369: Divide by Zero. The impact is: Divide by zero can lead to sudden crash of a software/service that tries to parse a .wav file.*

CVE-2019-1010315 : Le fix

```
 else if (!strncmp (dff_chunk_header.ckID, "DSD ", 4)) {

+    if (!config->num_channels) {
+        error_line ("%s is not a valid .DFF file!", infilename);
+        return WAVPACK_SOFT_ERROR;
+    }

     total_samples = dff_chunk_header.ckDataSize / config->num_channels
     break;
 }
```

Détecter les divisions par zéro

```
def diff_from_zero_checks =
  cpg.call(".*notEquals.*")
     .where(_.argument.filter(_.code == "0"))
     .argument.isIdentifier

def divisors =
  cpg.method.name("<operator>.division")
     .parameter.argument
     .whereNot(_.isLiteral).filter(_.argumentIndex == 2)

divisors.filter(_.reachableBy(diff_from_zero_checks).size == 0).
```

RCE dans VLC

Vulnérabilité dans VLC :

> *It is possible to trigger read or write buffer overflows with some crafted files or by a MITM attack on the automatic updater*

Vulnérabilité :

```
// pos et i_header_len contrôlés par l'attaquant
// i_packet_len contrôlé à 0xFFFFFFFF
int i_packet_len = scalar_number( pos, i_header_len );
...
// allocation de 0 bytes (0xFFFFFFFF + 0x1 = 0x0)
p_key->psz_username = (uint8_t*)malloc( i_packet_len + 1);
...
// copie de 0xFFFFFFFF bytes
memcpy( p_key->psz_username, pos, i_packet_len );
```

RCE dans VLC : détection avec Joern

```
// Source : appels à malloc avec de l'arithmétique
val src = cpg.method(".*malloc$").callIn.where(_.argument(1).arithmetic

// Appels à memcpy
cpg.method("(?i)memcpy").callIn.l.filter { memcpyCall =>
  memcpyCall
    // À un buffer correspondant
    .argument(1).reachableBy(src)
    .where(_.inAssignment.target.codeExact(memcpyCall.argument(1).code)
    // Mais pas la même taille
    .whereNot(_.argument(1).codeExact(memcpyCall.argument(3).code))
    .hasNext
}}.l
```

Quadros, Silva, 2023. Finding classes for exploiting Unsafe Reflection / Unchecked Class Instantiation vulnerabilities in Java with Joern

Contexte :

- une vulnérabilité permettant la création de classe arbitraire est trouvée
- on connaît les dépendances
- quelle classe instancier ?

```
jdbc:postgresql://127.0.0.1:5432/testdb?
  &socketFactory=CLASS&socketFactoryArg=ARGUMENT
```

Exploitation d'usages réflexifs non-sûrs en Java

On veut créer une classe qui :

- a un constructeur prenant une chaîne de caractère en argument
- font quelque chose d'intéressant (appel à `exec`, ...)

Implémentation : source et puits

Source : chaîne pouvant être passée à un constructeur :

```
val cons = cpg.method.isConstructor
              .signatureExact("void(java.lang.String)")
val source = cons.parameter.order(1)
```

Puits : appels à `Runtime.exec` :

```
val name = "java.lang.Runtime.exec:java.lang.Process(java.lang.String)"
val calls = cpg.method.fullNameExact(name).callIn
val sink = call.argument(1)
```

Implémentation : flots

```
sink.reachableByFlows(source).p
```

Implémentation : plus de puits

Méthodes permettant :

- Command Injection
- Expresion Language Injection (EL, OGNL, MVEL, SpEL, JEXL)
- Deserialization
- File read/write
- JDBC Driver Connect
- SSRF
- JNDI
- XML Beans
- XXE
- SQL Injection
- Code Loading (System.loadLibrary, java.net.URLClassLoader etc)
- Code Eval

Implémentation : validation dynamique

Pour chaque résultat :

- extraction du constructeur
- instanciation des valeurs pré-déterminées
- log pour analyse manuelle

Résultats

- Désérialisation non sécurisée : `net.sf.jasperreports.export.SimpleExporterInput` avec un chemin de fichier
- XXE : `org.apache.ibatis.parsing.XPathParser` avec du XML
- Fuite de fichier au travers de message d'erreur : `org.xbill.DNS.tools.jnamed` avec un chemin de fichier local à fuiter
- Injection de commande : `jline.UnixTerminal` avec `$(...)`
- SSRF : `com.itextpdf.text.pdf.codec.GifImage` avec une URL
- ...

Détection de pollution de prototypes

Fröberg, 2023. Detection of Prototype Pollution Using Joern.

- RQ1 : "Can Joern be used to correctly identify prototype pollution vulnerabilities?"
  - Résultats : environ 2/3 des vulnérabilités trouvées sur 80 paquets Node
- RQ2 : "What is the detection capability of the thesis' implemented query in Joern compared to queries in CodeQL when identifying prototype pollution vulnerabilities?"
  - précision inférieure, recall supérieur, effort moindre d'implémentation
- RQ3 : "What are the causes for Joern to wrongly identify a JavaScript program as vulnerable or nonvulnerable to prototype pollution?"
  - code dynamique, distinction tableau/objet, `module.exports`, bugs internes