Détection de flux de contrôle illégaux dans les Smartphones

Mariem Graa, Nora Cuppens-Bouhlahia, Frédéric Cuppens, Ana Cavalli

Détection de flux de contrôle illégaux dans les Smartphones

Mariem Graa[1,2] — Nora Cuppens-Boulahia[1] — Frédéric Cuppens[1] — Ana Cavalli[2]

[1] *Telecom-Bretagne*
2 Rue de la Chataigneraie,35576 Cesson Sevigne - France
{mariem.benabdallah,nora.cuppens,frederic.cuppens}@telecom-bretagne.eu

[2] *Telecom-SudParis*
9 Rue Charles Fourier, 91000 Evry - France
{mariem.graa, ana.cavalli}@it-sudparis.eu

RÉSUMÉ. *La sécurité dans les systèmes embarqués tels que les smartphones exige une protection des données privées manipulées par les applications tierces. Certains mécanismes utilisent des techniques d'analyse dynamique basées sur le « data-tainting » pour suivre les flux d'informations dans le programme. Mais ces techniques ne peuvent pas détecter les flux de contrôles qui utilisent des instructions conditionnelles pour transférer implicitement les informations. En particulier, les applications malveillantes peuvent contourner le système Android et obtenir des informations sensibles à travers les flux de contrôles. Nous proposons une amélioration de l'analyse dynamique qui propage la teinte tout au long des dépendances de contrôles en utilisant les données fournies par l'analyse statique dans les systèmes Android. Notre approche réussit à détecter des attaques de contrôle de flux sur les smartphones.*

ABSTRACT. *Security in embedded systems such as smartphones requires protection of private data manipulated by third-party applications. Many mechanisms use dynamic taint analysis techniques for tracking information flow in software. But these techniques cannot detect control flows that use conditionals to implicitly transfer information from objects to other objects. In particular, malicious applications can bypass Android system and get privacy sensitive information through control flows. We propose an enhancement of dynamic taint analysis that propagates taint along control dependencies by using the static analysis in embedded system such as Google Android operating system. Our approach allows the detection of control flow attacks on smartphones.*

MOTS-CLÉS : *Smartphones, Contrôle de flux, Tainting, Analyse statique, Analyse dynamique*

KEYWORDS: *Smartphones, Control flow, tainting, static analysis, dynamic analysis*

1. Introduction

Ces dernières années ont connu une augmentation de l'utilisation des systèmes embarqués tels que les Smartphones. D'après un récent rapport de Gartner (Egham, November 2010), 417 millions de téléphones mobiles ont été vendus au troisième trimestre de 2010, ce qui correspond à 35% d'augmentation par rapport à 2009. Les utilisateurs de Smartphones téléchargent des applications tierces pour rendre leurs téléphones plus utiles. Android Market annonce que le taux de téléchargement des applications tierces a dépassé 10 milliards d'applications en Décembre 2011 (Haselton, Decembre 2011). Ces applications peuvent être utilisées pour accéder et manipuler les données privées stockées dans les Smartphones. Selon une étude réalisée par Lookout Mobile Security, le nombre de malwares est en forte progression sur les plates-formes mobiles (Laporte, August 2011). Lookout Mobile Security prend l'exemple d'Android qui comptait 80 applications contenant du code malveillant en Janvier 2011. Ce chiffre a été multiplié par cinq en Juin 2011. Lookout Mobile Security estime que près de 500 000 personnes ont été victimes d'un malware sur Android au premier semestre de 2011. Dans l'étude présentée à la Conférence Black Hat, Daswani (Wilson, July 2011) a analysé le comportement de 10 000 applications Android et a montré que plus de 800 provoquent la fuite des données privées à un serveur non autorisé. Par conséquent, il est nécessaire de prévoir des mécanismes de sécurité adéquats pour contrôler la manipulation des données privées par des applications tierces. Plusieurs mécanismes sont utilisés pour protéger les données privées dans un système Android, telle que l'analyse dynamique qui est implémentée dans TaintDroid (Enck *et al.*, 2010). Le principe de l'analyse dynamique est d'associer une teinte aux données privées dans un système puis de propager cette teinte aux autres données qui en dépendent lors de l'exécution du programme pour suivre le flux d'information. Il existe deux types de flux d'information : les flux explicites et les flux implicites (flux de contrôle). Un exemple de flux explicite se produit lors d'une affectation $x = y$, où on observe un transfert explicite d'une valeur de x à y. Un exemple de flux de contrôle est illustré dans la Figure 1, où il n'y a pas de transfert direct de la valeur de a à b, mais lorsque le code est exécuté, b obtient la valeur de a.

```
1.boolean b = false;
2.boolean c = false;
3.if (!a)
4. c = true;
5.if (!c)
6.  b = true;
```

Figure 1. *Exemple de flux implicite.*

TaintDroid ne propage pas la teinte à travers les flux de contrôles ce qui provoque un problème d'under tainting : le processus de teintage tel que défini par Taintroid engendre des faux négatifs. Les applications malveillantes peuvent contourner un sys-

tème Android et obtenir des données privées en exploitant les flux de contrôles. Dans cet article, nous proposons une approche exhaustive et opérationnelle qui propage la teinte tout au long des flux de contrôles. Elle combine l'analyse statique et dynamique pour résoudre le problème d'under tainting. Notre approche détecte les attaques qui provoquent la fuite des données privées en exploitant les flux de contrôles au cours de l'exécution des applications Android. Nous allons, dans un premier temps, présenter dans la Section 2 des attaques exploitant des dépendances de contrôles que TaintDroid ne peut pas détecter. Dans la Section 3, nous analysons les travaux sur l'analyse statique et dynamique. Nous décrivons dans la Section 4 l'approche de TaintDroid. Nous présentons notre solution basée sur une approche hybride qui améliore la fonctionnalité de TaintDroid pour tracer les flux de contrôles dans la Section 5. Dans la section 6, nous montrons que notre approche détecte les attaques présentées dans la Section 2. Dans la section 7, nous présentons les résultats de test et d'évaluation de notre approche. La Section 8 est consacrée à une discussion portant sur les « faux positifs ». La Section 9 conclut l'article.

2. Attaques de contrôle de flux

Sarwar *et al.*(Sarwar *et al.*, 2013) présentent des attaques de contrôle de flux. Ces attaques visent le mécanisme de teintage dans TaintDroid. Le but de l'attaquant qui est le fournisseur de l'application malveillante est d'obfusquer le code et de tromper le processus de teintage dans TaintDroid pour qu'il ne teinte pas des données privées. Il joue sur la structuration du code (des flux) puisque TaintDroid ne propage pas la teinte dans les flux de contrôles. Sarwar *et al.* montrent expérimentalement le taux de réussite de ces attaques pour contourner la propagation de la teinte dans TaintDroid.

Algorithm 1 Attaque d'encodage

```
X ← Private_Data
for each x ∈ X do
  for each s ∈ AsciiTable do
    if (s == x) then
      Y ← Y + s
    end if
  end for
end for
Send_Network_Data(Y)
```

L'Algorithme 1 présente la première attaque. La variable X contient la donnée privée. L'attaquant utilise une boucle for pour parcourir les caractères de X et les comparer aux symboles de table ASCII. Les bons caractères sont concaténés dans la chaine de caractère Y. A la fin de cette boucle l'attaquant réussit à savoir la valeur de la donnée privée stockée dans Y. Comme TaintDroid ne propage pas la teinte dans les flux de contrôles, la variable Y n'est pas teintée. Elle est envoyée à travers le réseau

sans être détectée.

Algorithm 2 Attaque de compteur

$X \leftarrow Private\_Data$
for each $x \in X$ **do**
  $n \leftarrow CharToInt(x)$
  $y \leftarrow 0$
  for $i = 0$ to n **do**
    $y \leftarrow y + 1$
  end for
  $Y \leftarrow Y + IntToChar(y)$
end for
$Send\_Network\_Data(Y)$

L'Algorithme 2 présente la deuxième attaque. La donnée privée est affectée à la variable X. L'attaquant utilise de l'encodage de code pour convertir chaque caractère de X en un entier. En utilisant un compteur, il réussit à savoir la valeur de cet entier. Ensuite, il reprend la technique d'encodage pour convertir cet entier en un caractère. Les caractères obtenus sont concaténés dans Y. A la fin de l'exécution de cet algorithme la variable Y contient la donnée privée mais elle n'est pas teintée car TaintDroid ne détecte pas les flux de contrôles. Elle est envoyée à travers le réseau sans être détectée.

Algorithm 3 Attaque d'exception

$X \leftarrow Private\_Data$
for each $x \in X$ **do**
  $n \leftarrow CharToInt(x)$
  $y \leftarrow 0$
  while $y < n$ **do**
    $Try\{$
    $Throw\_New\_Exception()\}$
    $Catch(Exception\ e)\{$
    $y \leftarrow y + 1\}$
  end while
  $Y \leftarrow Y + IntToChar(y)$
end for
$Send\_Network\_Data(Y)$

L'Algorithme 3 présente une attaque basée sur les exceptions. La variable n contient un entier qui correspond à la conversion d'un caractère de la donnée privée. L'attaquant lance une exception n fois.
Il traite l'exception dans le bloc "catch" en incrémentant la variable y pour atteindre la valeur de n. La conversion de y en caractère permet de savoir un caractère de la donnée privée. En concaténant tous les caractères trouvés, l'attaquant réussit à connaitre la valeur de la donnée privée. TaintDroid ne propage pas la teinte dans les exceptions

utilisées dans les flux de contrôles. Par conséquent, l'envoi de la variable Y contenant la donnée privée n'est pas détecté par TaintDroid. Nous présentons dans la section suivante les approches existantes utilisant l'analyse statique et dynamique qui peuvent être utilisées pour détecter des attaques définies par les dépendances de contrôles.

3. Travaux de recherche associés

Data Tainting permet de tracer la propagation des données dans un système. Le principe de ce mécanisme est de teinter (associer un tag) les données dans un programme et de propager la teinte aux objets dépendants. Il est utilisé pour la détection de vulnérabilités, la protection des données sensibles et plus récemment, pour l'analyse de malware. Une vulnérabilité est détectée lorsqu'une donnée teintée est utilisée dans un « Taint Sink ». Le data tainting est implémenté dans les interpréteurs (Wall *et al.*, 2000),(Hunt et Thomas, 2000) pour suivre les données sensibles. Il est utilisé pour analyser dynamiquement le code binaire (Newsome et Song, 2005),(Cheng *et al.*, 2006),(Qin *et al.*, 2006),(Yin *et al.*, 2007) en l'instrumentant pour suivre la propagation de la teinte. Ainsi, ce mécanisme dégrade les performances (temps d'exécution) du système ce qui ne favorise pas leur utilisation dans des applications exécutées en temps réel. Plusieurs approches étudient la protection des données personnelles sur les smartphones. Les approches basées sur le contrôle d'accès (Enck *et al.*, 2009 ; Nauman *et al.*, 2010 ; Conti *et al.*, 2011 ; Bugiel *et al.*, 2011 ; Ongtang *et al.*, 2010) assurent que seules les applications qui ont les droits nécessaires peuvent accéder aux données privées. Mais, ces approches n'assurent pas la protection des données de bout en bout. Elles sont complétées par les approches qui contrôlent le flux des données. TaintDroid (Enck *et al.*, 2010) implémente l'analyse dynamique pour suivre les flux de données dans les applications en temps réel. Enck *et al.* s'inspirent des travaux précédents, mais ils adressent des défis différents spécifiques aux smartphones comme les ressources limitées. AppFence (Hornyack *et al.*, 2011) est une extension de Taintdroid qui détecte et bloque l'envoi des données privées en dehors du système. Ces approches permettent de suivre seulement les flux explicites et ils ne détectent pas les flux de contrôles. Ainsi, ils ne peuvent pas détecter les attaques liées aux dépendances de contrôles qui visent la vie privée des utilisateurs. Cavallaro *et all.* (Cavallaro *et al.*, 2008) décrivent les techniques d'évasion contre l'analyse dynamique des flux d'informations. Ces attaques d'évasion utilisent les dépendances de contrôles et provoquent la fuite des données privées. Cavallaro *et al.* pensent qu'il est nécessaire de raisonner sur les affectations des variables dans les structures conditionnelles. Nous avons suivi ce raisonnement dans l'implémentation des règles qui définissent la propagation de la teinte. Certaines approches existent dans la littérature pour suivre les flux de contrôles (Egele *et al.*, 2007), (Song *et al.*, 2008), (Kang *et al.*, 2011), (Nair *et al.*, 2008). Elles combinent l'analyse statique et dynamique pour identifier correctement les flux implicites et pour détecter la fuite des données sensibles. DTA ++ (Kang *et al.*, 2011) étend l'analyse dynamique pour suivre les flux de contrôles. Cependant, DTA ++ est évaluée uniquement sur les applications bénignes et il n'est pas testé sur les programmes malveillants. En outre, ces approches ne sont

pas implémentées dans les systèmes embarqués tel que les smartphones. L'analyse statique implémentée dans les smartphones (Egele *et al.*, 2011 ; Chin *et al.*, 2011 ; Fuchs *et al.*, 2009) permet de détecter les fuites de données, mais elle ne peut pas capturer toutes les configurations durant la phase d'exécution. Nous nous sommes inspirés de ces travaux, mais nous avons utilisé une approche qui combine l'analyse statique et l'analyse dynamique pour détecter les attaques exploitant les dépendances de contrôles dans le système Android. Fenton (Fenton, 1974) a proposé une Machine "Data Mark", un modèle abstrait, pour gérer les flux de contrôles. Il donne une description formelle de son modèle et une preuve de correction en termes de flux d'information. Aries (Brown et Knight Jr, 2001) interdit l'écriture à un emplacement particulier dans la branche conditionnelle lorsque la classe de sécurité associée à cet emplacement est égale ou moins restrictive que celles de compteur de programme. L'approche d'Aries utilise uniquement les classes de sécurité de niveau haut et bas. Denning (Denning, 1975) améliore le mécanisme défini par Fenton en terme de temps d'exécution avec un mécanisme se déclenchant pendant la compilation pour détecter tous les flux de contrôles. Il insère des instructions supplémentaires si la branche est prise ou pas pour déterminer la classe de sécurité modifiée afin de refléter le flux d'informations. Nous nous inspirons de l'approche de Denning, mais nous avons formellement défini un ensemble de règles de propagation de la teinte afin d'éviter les attaques exploitant les dépendances de contrôles. Nous décrivons dans la Section suivante plus en détail l'approche de TaintDroid.

4. Système de contrôle de flux : Taintroid

Les applications tierces installées sur un smartphone peuvent extraire des données privées de l'utilisateur. TaintDroid est une extension de la plateforme Android. Il est implémenté dans la machine virtuelle de smartphone.

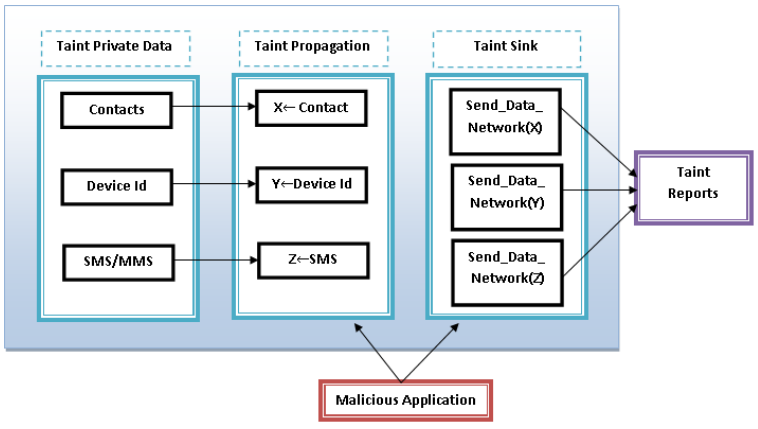


Figure 2. *Processus de TaintDroid*

TaintDroid utilise le mécanisme de data tainting et l'analyse dynamique pour suivre les flux explicites en temps réel et pour contrôler la manipulation des données privées par les applications tierces. Le processus de TaintDroid est présenté dans la Figure 2. D'abord, il associe une teinte aux données privées. Ensuite, il suit la propagation des données teintées. Enfin, il détecte une vulnérabilité si une donnée teintée est utilisée dans un emplacement sensible (taint sink) qui permet d'envoyer la donnée en dehors du système. L'inconvénient de TaintDroid est qu'il ne détecte pas les flux de contrôles. Donc il ne peut pas détecter les attaques exploitant les dépendances de contrôles. Nous décrivons notre approche plus en détail dans la section suivante.

5. Détection des flux de contrôles dans les Smartphones

Comme précisé dans la Section 4 TaintDroid ne propage pas la teinte à travers les flux de contrôles ce qui cause un problème d'under tainting. Dans cette section, nous commençons par donner une spécification formelle du problème d'under tainting. Ensuite, nous présentons notre solution formelle basée sur deux règles qui définissent la politique de teintage. Enfin, nous présentons notre solution technique qui propage la teinte tout au long des dépendances de contrôles en utilisant les deux règles de propagation pour résoudre le problème d'under tainting.

5.1. *Spécification formelle du problème d'under tainting*

Denning (Denning, 1976) définit un modèle de flux d'informations comme suit :

$$FM =< N, P, SC, \oplus, \rightarrow > .$$

– N est un ensemble d'objets de stockage logique (fichiers, variables,...).

– P est un ensemble de processus qui sont exécutés par les agents responsables de tous les flux d'informations.

– SC est un ensemble de classes de sécurité qui sont affectées aux objets de N. SC est un ensemble fini qui a une limite inférieure L associée aux objets de N.

– l'opérateur de combinaison de classe « $\oplus$ » spécifie la classe résultat de n'importe quelle fonction binaire ayant comme opérande des classes de sécurité.

– une relation de flux « $\rightarrow$ » entre les paires de sécurité classes A et B signifie que « les informations en classe A sont autorisées à circuler vers la classe B ». Un modèle de flux FM est sûr si et seulement si l'exécution d'une séquence d'opérations ne peut pas produire un flux qui viole la relation « $\rightarrow$ ».

Nous spécifions formellement le problème d'under tainting en utilisant le modèle de flux d'informations de Denning. Mais, nous attribuons une teinte aux objets au lieu d'assigner des classes de sécurité. Ainsi, l'opérateur de combinaison de classe « $\oplus$ » est utilisé dans notre spécification formelle pour combiner les teintes des objets.

Définition syntaxique des connecteurs $\{\Rightarrow, \rightarrow, \leftarrow, \oplus\}$ **:** Nous utilisons la syntaxe suivante pour spécifier formellement le problème d'under tainting : A et B sont deux formules logiques et x et y sont deux variables.

– $A \Rightarrow B$: si A alors B

– $x \rightarrow y$: L'information circule de l'object x à l'object y

– $x \leftarrow y$: la valeur de y est affectée à x

– $Taint(x) \oplus Taint(y)$: spécifie la teinte résultat de combinaisons des teintes.

Définition sémantique de connecteurs $\{\rightarrow, \leftarrow, \oplus\}$ **:**

– Le connecteur $\rightarrow$ est réflexif : Si x est une variable alors $x \rightarrow x$.

– Le connecteur $\rightarrow$ est transitif : x, y et z sont trois variables, si $(x \rightarrow y) \wedge (y \rightarrow z)$ alors $x \rightarrow z$.

– Le connecteur $\leftarrow$ est réflexif : Si x est une variable alors $x \leftarrow x$.

– Le connecteur $\leftarrow$ est transitif : x, y et z sont trois variables, si $(x \leftarrow y) \wedge (y \leftarrow z)$ alors $x \leftarrow z$.

– Les connecteurs $\rightarrow$ et $\leftarrow$ ne sont pas symétriques.

– La relation $\oplus$ est commutative : $Taint(x) \oplus Taint(y) = Taint(y) \oplus Taint(x)$

– La relation $\oplus$ est associative : $Taint(x) \oplus (Taint(y) \oplus Taint(z)) = (Taint(x) \oplus Taint(y)) \oplus Taint(z)$

Definition. Une situation d'under tainting (sous teintage) se produit lorsque x dépend d'une $condition$, on affecte à x une valeur dans la branche conditionnelle et $condition$ est teintée mais x n'est pas teinté. Formellement,

$$\begin{aligned} \textit{Affectation}(x, y) \wedge Dependance(x, condition) \\ \wedge Teinte(condition) \wedge \neg Teinte(x) \end{aligned} \tag{1}$$

où :

– *Affectation*(x, y) affecte à x la valeur de y.

$$\textit{Affectation}(x, y) \stackrel{def}{\equiv} (x \leftarrow y)$$

– $Dependency(x, condition)$ definit un flux d'information de la $condition$ à x si x depend de la $condition$.

$$Dependance(x, condition) \stackrel{def}{\equiv} (condition \rightarrow x)$$

5.2. *Solution formelle de l'under tainting dans les smartphones*

Pour résoudre le problème d'under tainting, nous proposons un ensemble de règles qui définit la politique de teintage permettant de détecter les attaques exploitant les dépendances de contrôles. Grâce à ces règles, toutes les variables auxquelles une valeur

est assignée dans la structure conditionnelle sont teintées que cette branche soit prise ou pas. Nous considérons que le $Contexte\_Teinte$ est la teinte de la $condition$.

– règle 1 : si la valeur de x est modifiée, x dépend de la $condition$ et la branche est prise, nous appliquons la règle suivante pour teinter x.

$$\frac{\textit{Modifier}(x) \wedge Dependance(x, condition) \wedge Branche\_Prise(br, inst\_cond)}{Teinte(x) \leftarrow Contexte\_Teinte \oplus Teinte(inst\_flux\_explicite)}$$

où : Le predicat $Branche\_Prise(br, inst\_cond)$ spécifie que la branche br dans l'instruction conditionnelle est exécutée, et donc un flux explicite qui contient x est exécuté.

Modifier (x, inst_flux_explicite) associe à x le résultat de flux explicite.

$$\textit{Modifier}(x) \stackrel{def}{\equiv} \quad \textit{Affectation(x, inst\_flux\_explicite)}$$

– Règle 2 : si la valeur de y est affectée à x, x dépend de la $condition$ et la branche br dans l'instruction conditionnelle n'est pas prise (x ne dépend que du flux implicite et ne dépend pas du flux explicite), nous appliquons la règle suivante pour teinter x.

$$\frac{\textit{Affectation}(x, y) \wedge Dependance(x, condition) \wedge \neg Branche\_Prise(br, inst\_cond)}{Teinte(x) \leftarrow Teinte(x) \oplus Contexte\_Teinte}$$

Dans ce papier, nous ne reprenons pas la preuve de complétude de ces règles qui est accessible dans (Graa *et al.*, 2013). Dans (Graa *et al.*, 2013), nous avons prouvé la complétude de ces règles. Aussi, nous avons fourni un algorithme correct et complet utilisant ces règles qui permet de résoudre le problème d'under tainting.

5.3. *Solution technique de l'under tainting dans les smartphones*

Pour résoudre le problème d'under tainting, nous sommes intervenu au niveau de l'architecture de TaintDroid. Nous avons implémenté une approche hybride qui combine l'analyse statique et l'analyse dynamique. Nous avons défini et implémenté un module « implicit flow tracking » dans le vérificateur de code Dex de la machine virtuelle Dalvik. Ce module effectue l'analyse statique et vérifie les instructions au moment de l'installation des applications Android. Nous modifions l'interprèteur de la machine virtuelle Dalvik et nous ajoutons les deux règles présentées dans la Section 5.2 pour propager la teinte tout au long des flux de contrôles.

5.3.1. *Analyse statique du Code dex*

Nous effectuons une analyse statique au moment de l'installation des applications Android. Cette analyse utilise des graphes de flot de contrôle qui sont composés des

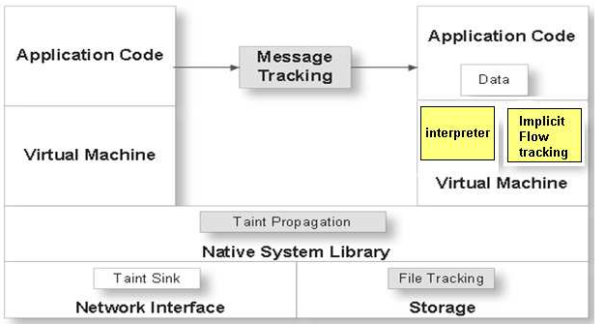


Figure 3. *Architecture modifiée pour gérer les flux implicites dans le système TaintDroid.*

blocs de base et des arêtes. Un bloc de base représente une instruction de contrôle. Nous utilisons « post dominator » pour déterminer la dépendance des différents blocs au bloc de la condition. Les graphes de flot de contrôle sont stockés sous le format graphviz (Research,) dans le dossier de données du smartphone. Les tailles des graphes de flot de contrôle que nous avons obtenus dans nos différents tests sont de l'ordre de 1200 octets.

5.3.2. *Analyse Dynamique des Applications Android*

Nous implémentons l'analyse dynamique au moment de l'exécution en utilisant les informations fournies par l'analyse statique. Nous attribuons un context_taint à chaque bloc de base. Le $Contexte\_Teinte$ contient la teinte de la condition dont dépend le bloc. Nous commençons par les branches qui ne sont pas prises. Nous utilisons l'analyse statique pour déterminer le type et le nombre d'instructions dans ces branches. Ensuite, nous forçons le processeur à exécuter ces instructions et de teinter les variables auxquelles une valeur est affectée en utilisant la deuxième règle de propagation de la teinte. Nous attribuons seulement une teinte aux variables et nous ne modifions pas leurs valeurs. Enfin, nous restaurons le compteur de programme pour pointer vers la première instruction dans la branche qui est prise. Nous attribuons une teinte aux variables modifiées dans cette branche en utilisant la première règle de propagation de la teinte. Nous implémentons les deux règles qui définissent la politique de teintage dans le module « Taint Propagation » de Taintdroid. L'architecture modifiée pour gérer les flux implicites dans le système TaintDroid est illustrée dans la Figure 3.

6. Détection des attaques de contrôle de flux

Nous avons implémenté et testé les trois attaques exploitant les dépendances de contrôles présentées dans la Section 2 en utilisant un smartphone « Nexus One » ayant un système d'exploitation Android 2.3 que nous avons modifié pour suivre les flux

```
W/dalvikvm( 1209): TaintLog: OSNetworkSystem.write(10.35.131.42) received data with tag
 0x2 data=[??
          Mariem Graa]
```

(a)

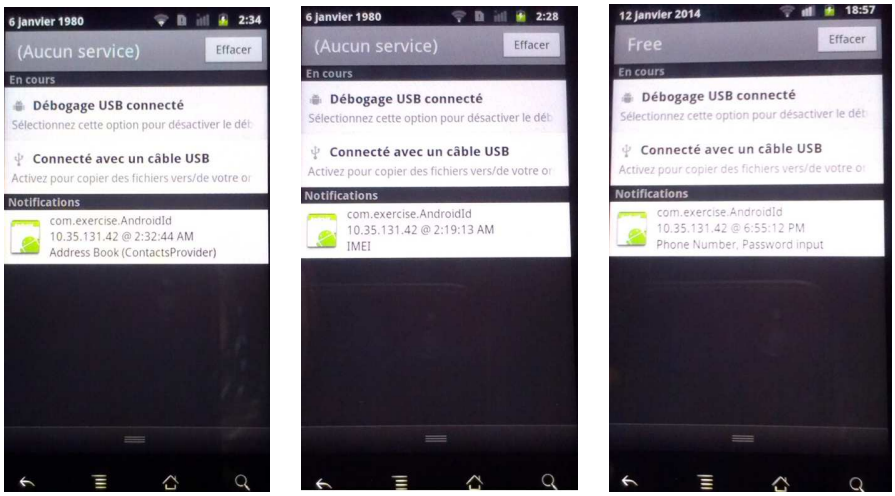
```
W/dalvikvm(  712): TaintLog: OSNetworkSystem.write(10.35.131.42) received data with
 tag 0x400 data=[??354957033679070]
```

(b)

```
W/dalvikvm(  488): TaintLog: OSNetworkSystem.write(10.35.131.42) received data with tag
 0x10008 data=[??
W/dalvikvm(  488): 3627890380]
```

(b)

Figure 4. *Fichier Log des attaques*



(a) (b) (c)

Figure 5. *Notification signalant la fuite des données sensibles*

de contrôles. Nous utilisons l'outil Traceview pour évaluer les performances de ces attaques.

Considérons la première attaque (voir l'Algorithme 1). Nous avons remplacé la donnée privée par le nom du contact de l'utilisateur (« Graa Mariem »). Il compare les caractères de cette donnée avec les symboles de la table Ascii dans la deuxième boucle. La teinte du contact de l'utilisateur est $((u4)0 \times 00000002)$.

La variable x est teintée car elle appartient à la chaîne de caractères X qui est teintée. Ainsi, la condition dans l'instruction $if(x == TabAsc[j])$ est teintée. Notre système propage la teinte dans le flux de contrôle. En appliquant la première règle, Y est teinté et $Taint(Y) = Taint(x == TabAsc[j]) \oplus Taint(Y + TabAsc[j])$. Nous pouvons voir dans le fichier log (Figure 4(a)) que Y est teinté et que la teinte de Y

est égale à la teinte du nom de contact de l'utilisateur. Une notification apparaît (voir Figure 5(a)) pour prévenir l'utilisateur de la fuite de Y qui contient la valeur de la donnée privée. Le premier algorithme est exécuté dans $88ms$ en utilisant Taintdroid modifié pour suivre les flux de contrôles et $36ms$ dans un Android non modifié.

Au niveau de la deuxième attaque (voir l'Algorithme 2), l'attaquant essaye d'obtenir une information secrète X qui est le numéro IMEI du smartphone. La teinte de l'IMEI est $((u4)0 \times 00000400)$. La variable x est teintée car elle appartient à la chaine de caractère X qui est teintée. Le résultat n de la conversion de x en entier est teinté. Ainsi, la condition $(i = 0\ to\ n)$ est teintée. En appliquant la première règle, y est teinté et $Taint(y) = Taint(i = 0\ to\ n) \oplus Taint(y + 1)$. Dans la première boucle, la condition $x \in X$ est teintée. Nous appliquons la première règle, Y est teinté et $Taint(Y) = Taint(x \in X) \oplus Taint(Y + (char)y)$. Ce résultat apparait dans le fichier log illustré dans la Figure 4(b). La fuite de la donnée privée est aussi détectée par notre approche (voir la notification dans Figure 5(b)). Le deuxième algorithme est exécuté dans $101ms$ en utilisant Taintdroid modifié pour suivre les flux de contrôles et de $20ms$ dans un Android non modifié. Le temps d'exécution dans notre approche est plus important parce qu'il inclut le temps de la propagation de la teinte dans les flux de contrôles.

L'attaquant exploite une exception pour lancer la troisième attaque (voir l'Algorithme 3) qui provoque la fuite des données sensibles (numéro de téléphone). Nous avons choisi la division par zéro comme exception arithmétique. Cette exception est teintée et dépend de la condition de la boucle $while$ qui est $y < n$. Cette condition est teintée parce que la variable n qui correspond à la conversion d'un caractère dans $phone\_number$ est teintée. Comme TaintDroid, ne teinte pas les exceptions, nous définissons une teinte d'exception ($Taint\_Exception = ((u4)0 \times 00010000)$). Ensuite, nous propageons la teinte de l'exception dans le bloc catch. Nous appliquons la première règle pour teinter y. On obtient $Taint(y) = Taint(exception) \oplus Taint(y+1)$. Enfin, la chaîne de caractère Y qui contient la donnée privée est teintée et $Taint(Y) = Taint(x \in X) \oplus Taint(Y + (char)y)$. Dans le fichier log illustré dans la Figure 4(c), on peut voir que la teinte de Y est la combinaison de la teinte de l'exception $(((u4)0 \times 00010000))$ et la teinte du numéro de téléphone $(((u4)0 \times 00000008))$. Un message d'alerte apparaît au moment de l'envoi de la donnée sensible (voir la notification dans la Figure 5(c)). L'exécution du troisième algorithme nécessite $1437ms$ en utilisant Taintdroid modifié pour suivre les flux de contrôles et $1385ms$ dans Android non modifié. Cette différence est due à la propagation de la teinte dans le flux de contrôles.

7. Evaluation

Nous avons utilisé CaffeineMark (CORPORATION,) afin de déterminer le « java microbenchmark ». Le premier algorithme présente un score global de 3486 Java instructions exécutées par seconde en utilisant un Android non modifié, et 2893 Java instructions exécutées par seconde en utilisant notre approche. Le deuxième algo-

rithme présente un score global de 3465 en utilisant un Android non modifié et 2893 en utilisant notre approche.

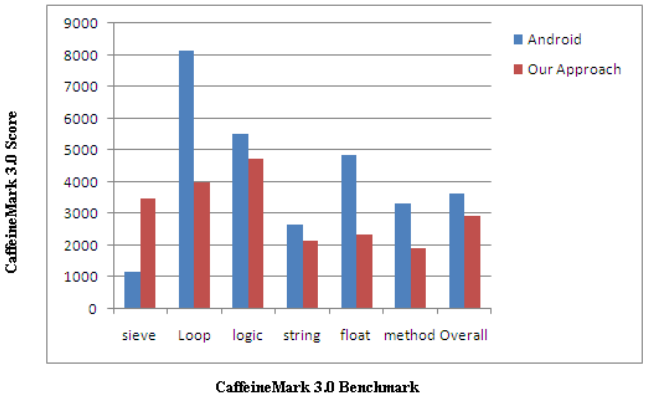


Figure 6. *Microbenchmark de java overhead*

Le troisième algorithme présente un score global de 4241 en utilisant un Android non modifié et 3440 en utilisant approche. Nous avons tester d'autres algorithmes plus complexes. La Figure 6 présente les résultats obtenus. Nous propageons la teinte dans les branches conditionnelles en particulier dans la boucle for et nous ajoutons des instructions dans le processeur pour résoudre le problème d'under tainting ce qui explique le temps d'exécution élevé au niveau de « loop benchmark ». Nous associons une teinte aux résultats des opérations arithmétiques dans les flux explicites et les flux de contrôles. Ainsi, les opérations arithmétiques présentent un temps d'exécution élevé. La différence de « string benchmark » entre un système Android non modifié et notre approche est due à la mémoire supplémentaire requise dans la propagation de la teinte dans les objets string. Nous constatons que le système Android non modifié présente un score global de 3625 Java instructions exécutées par seconde. Notre approche présente un score global de 2937 Java instructions exécutées par seconde. Par conséquent, notre approche crée un overhead de 19% du à la propagation de la teinte dans les flux de contrôles. Cette overhead semble acceptable en comparaison à celui créé par TaintDroid qui est de 14%.

8. Discussion

Dans notre approche, nous associons une teinte à toutes les variables dans la structure conditionnelle. Cela peut causer un problème de faux positifs. Le problème a été abordé dans (Kang *et al.*, 2011) et (Bao *et al.*, 2010). Cependant, ces approches ne proposent pas de solutions. Kang *et al.* (Kang *et al.*, 2011) utilisent une technique de diagnostic pour sélectionner les branches qui pourraient être responsables de l'under tainting et propagent la teinte que seulement tout au long de ces branches afin de

réduire les faux positifs. Cependant DTA++ produit encore des faux positifs. Bao *et al.* (Bao *et al.*, 2010) ne considèrent pas toutes les dépendances de contrôles pour réduire le nombre de faux positifs. Leur mise en oeuvre donne des résultats similaires à DTA++ dans de nombreux cas, et se fonde sur la syntaxe d'une expression de comparaison. En revanche, DTA++ utilise une condition plus générale et plus précise au niveau sémantique, implémentée à l'aide d'une exécution symbolique. Notre approche peut produire des faux positifs, mais elle offre plus de sécurité car toutes les données privées sont teintées. Ainsi, on ne peut pas avoir une fuite des informations sensibles. Nous nous sommes intéressés à résoudre le problème d'under tainting car nous considérons que les faux négatifs sont plus dangereux que les faux positifs étant donné que les faux négatifs peuvent créer des failles de sécurité. Nous avons l'intention de réduire les faux positifs en appliquant des règles expertes. Ce qui permet de réduire aussi les temps d'éxecution trouvés dans la section précédente.

9. Conclusion

Afin de protéger les smartphones des attaques exploitant les dépendances de contrôles, nous avons proposé une approche formelle et technique qui combine l'analyse statique et dynamique. Dans cet article, nous avons présenté des attaques utilisant les dépendances de contrôles qui provoquent la fuite des données sensibles. Nous avons spécifié formellement deux règles qui définissent la politique de teintage pour détecter ces attaques exploitant les dépendances de contrôles. Nous avons montré que notre approche réussit à détecter ces attaques. Ainsi, les applications malveillantes ne peuvent pas contourner le système Android et obtenir des informations sensibles en exploitant les flux de contrôles. Nous avons fait des tests d'évaluation de performance de notre approche qui crée un overhead de 19%. Nous planifions de tester des structures conditionnelles plus complexes (if imbriqué, switch, ...) et d'autres types d'attaques exploitant les dépendances de contrôles. Nous planifions aussi d'affiner notre approche pour réduire le nombre de fausses alarmes.

10. Bibliographie

Bao T., Zheng Y., Lin Z., Zhang X., Xu D., « Strict control dependence and its effect on dynamic information flow analyses », *Proceedings of the 19th international symposium on Software testing and analysis*, ACM, 2010, p. 13–24.

Brown J., Knight Jr T., « A minimal trusted computing base for dynamically ensuring secure information flow », *Project Aries TM-015 (November 2001)*, , 2001.

Bugiel S., Davi L., Dmitrienko A., Heuser S., Sadeghi A.-R., Shastry B., « Practical and lightweight domain isolation on android », *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, ACM, 2011, p. 51–62.

Cavallaro L., Saxena P., Sekar R., « On the limits of information flow techniques for malware analysis and containment », *Detection of Intrusions and Malware, and Vulnerability Assessment*, p. 143–163, Springer, 2008.

Cheng W., Zhao Q., Yu B., Hiroshige S., « Tainttrace : Efficient flow tracing with dynamic binary rewriting », *ISCC'06. Proceedings. 11th IEEE Symposium on*, IEEE, 2006, p. 749–754.

Chin E., Felt A. P., Greenwood K., Wagner D., « Analyzing inter-application communication in Android », *Proceedings of the 9th international conference on Mobile systems, applications, and services*, ACM, 2011, p. 239–252.

Conti M., Nguyen V., Crispo B., « CRePE : Context-related policy enforcement for Android », *Information Security*, , 2011, p. 331–345, Springer.

CORPORATION P. S., « CaffeineMark 3.0 ».

Denning D., « Secure information flow in computer systems », PhD thesis, Purdue University, 1975.

Denning D., « A lattice model of secure information flow », *Communications of the ACM*, vol. 19, n° 5, 1976, p. 236–243, ACM.

Egele M., Kruegel C., Kirda E., Yin H., Song D., « Dynamic spyware analysis », *Usenix Annual Technical Conference*, 2007.

Egele M., Kruegel C., Kirda E., Vigna G., « PiOS : Detecting privacy leaks in iOS applications », *Proceedings of the Network and Distributed System Security Symposium*, 2011.

Egham U., « Gartner Says Worldwide Mobile Phone Sales Grew 35 Percent in Third Quarter 2010 : Smartphone Sales Increased 96 Percent », November 2010.

Enck W., Ongtang M., McDaniel P., « On lightweight mobile phone application certification », *Proceedings of the 16th ACM conference on Computer and communications security*, ACM, 2009, p. 235–245.

Enck W., Gilbert P., Chun B., Cox L., Jung J., McDaniel P., Sheth A., « TaintDroid : An information-flow tracking system for realtime privacy monitoring on smartphones », *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, USENIX Association, 2010, p. 1–6.

Fenton J., « Memoryless subsystem », *Computer Journal*, vol. 17, n° 2, 1974, p. 143–147.

Fuchs A. P., Chaudhuri A., Foster J. S., « SCanDroid : Automated security certification of Android applications », *Manuscript, Univ. of Maryland, http ://www.cs.umd.edu/˜ avik/projects/scandroidascaa*, , 2009, Citeseer.

Graa M., Cuppens-Boulahia N., Cuppens F., Cavalli A., « Formal Characterization of Illegal Control Flow in Android System », *Signal Image Technology & Internet Systems*, IEEE, 2013.

Haselton, « Android Market surpasses 10 billion app downloads, http ://bgr.com/2011/12/06/android-market-surpasses-10-billion-app-downloads-google-kicks-off-0-10-app-sale/ », Decembre 2011.

Hornyack P., Han S., Jung J., Schechter S., Wetherall D., « These aren't the droids you're looking for : retrofitting android to protect data from imperious applications », *Proceedings of the 18th ACM conference on Computer and communications security*, ACM, 2011, p. 639–652.

Hunt A., Thomas D., « Programming Ruby : The Pragmatic Programmer's Guide », *New York : Addison-Wesley Professional.*, vol. 2, 2000.

Kang M., McCamant S., Poosankam P., Song D., « DTA++ : Dynamic taint analysis with targeted control-flow propagation », *Proc. of the 18th Annual Network and Distributed*

System Security Symp. San Diego, CA, 2011.

Laporte, « Les malwares débarquent sur les smartphones, http ://www.igen.fr/iphone/les-malwares-debarquent-sur-les-smartphones-55012% guilf, August 2011.

Nair S., Simpson P., Crispo B., Tanenbaum A., « A virtual machine based information flow control system for policy enforcement », *Electronic Notes in Theoretical Computer Science*, vol. 197, n° 1, 2008, p. 3–16, Elsevier.

Nauman M., Khan S., Zhang X., « Apex : extending android permission model and enforcement with user-defined runtime constraints », *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, ACM, 2010, p. 328–332.

Newsome J., Song D., « Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software », Citeseer, 2005.

Ongtang M., Butler K., McDaniel P., « Porscha : Policy oriented secure content handling in Android », *Proceedings of the 26th Annual Computer Security Applications Conference*, ACM, 2010, p. 221–230.

Qin F., Wang C., Li Z., Kim H., Zhou Y., Wu Y., « Lift : A low-overhead practical information flow tracking system for detecting security attacks », *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, IEEE Computer Society, 2006, p. 135–148.

Research A., « Graphviz,http ://www.graphviz.org/ ».

Sarwar G., Mehani O., Boreli R., Kaafar M. A., « On the Effectiveness of Dynamic Taint Analysis for Protecting Against Private Information Leaks on Android-based Devices », 2013.

Song D., Brumley D., Yin H., Caballero J., Jager I., Kang M., Liang Z., Newsome J., Poosankam P., Saxena P., « BitBlaze : A new approach to computer security via binary analysis », *Information Systems Security*, , 2008, p. 1–25, Springer.

Wall L., Christiansen T., Orwant J., *Programming perl*, O'Reilly Media, 2000.

Wilson T., « Many Android Apps Leaking Private Information », July 2011.

Yin H., Song D., Egele M., Kruegel C., Kirda E., « Panorama : capturing system-wide information flow for malware detection and analysis », *Proceedings of the 14th ACM Conference on Computer and Communications Security*, ACM, 2007, p. 116–127.