Protection des systèmes face aux attaques par fuzzing

Catégorie: protection-défense-détection

Léopold Ouairy, Hélène Le-Bouder, Jean-Louis Lanet,
{leopold.ouairy,jean-louis.lanet}@inria.fr, helene.le-bouder@imt-atlantique.fr

INRIA, IMT

Abstract. Le but de ce projet est de protéger les applets *Java Card* contre les attaques par fuzzing. Ces dernières permettent à un attaquant d'accéder à des ressources restreintes en exploitant une mauvaise implémentation d'une machine à états. Pour ce faire, nous créons un outil *ChuckyJava*. Il vise à détecter les vérifications manquantes (*missing-checks*) par machine learning non supervisé. Il transforme les fonctions en vecteurs, les comparent et est capable de déterminer si une fonction est vulnérable. Nous exécutons *ChuckyJava* sur cinq applets implémentant la spécification d'*OpenPGP*. Nous présentons deux vulnérabilités et un problème d'optimisation.

Keywords: Machine Learning non supervisé, K-Nearest-Neighbors, Attaques par fuzzing

1 Introduction

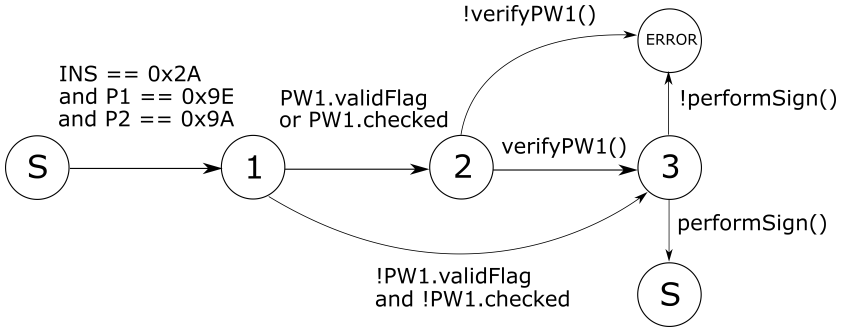
Les attaques par fuzzing consistent à envoyer des messages à un programme afin de tester au maximum son flot de contrôle. Cela permet à un attaquant de détecter une déviation dans le fonctionnement du programme. Cela peut mener à un changement d'état du programme qui n'était pas autorisé par les développeurs. De plus, en cas de présence d'un manque de vérification d'un paramètre, un attaquant peut accéder à des ressources normalement interdites. En plus d'autres attaques, le fuzzing peut être utilisé pour tester un produit avant son déploiement afin de tester les mauvaises implémentations de spécifications.
Afin de protéger les applets des attaques par fuzzing, nous créons l'outil *ChuckyJava*. Il est basé sur une version améliorée de *Chucky*[1, 2]. Il repose sur une technique de machine-learning non supervisé. Nous l'adaptons afin qu'il puisse fonctionner sur des applets *Java Card*. Nous voulons ainsi analyser la présence de vulnérabilités dans ces codes sources. Ensuite, nous présentons nos résultats sur des applets implémentant *OpenPGP*. Nous présentons deux améliorations possibles de code ainsi que deux vulnérabilités. Enfin, nous concluons sur nos résultats et exposons nos travaux à venir.
La section 2 présente le contexte et l'état de l'art. La description de *Chucky* est effectuée en section 3. L'adaptation et les modifications sont présentées en section 4. La section 5 présente les résultats obtenus. Nous exposons quatre limitations en section 6. Enfin, nous concluons en section 7.

2 Contexte et état de l'art

2.1 Contexte

Un programme conçu comme une machine à états accepte seulement un ensemble de commandes spécifique à chaque état. Ces commandes permettent au programme de changer son état actuel pour un autre. Une spécification clarifie à la fois les états et les transitions qui sont à implémenter dans un programme. *OpenPGP* est une version ouverte du standard de chiffrement *Pretty Good Privacy* (*PGP*). Fig. 1 montre une partie de la machine à état pour la commande *COMPUTE_DIGITAL_SIGNATURE* d'*OpenPGP*. Elle permet ou non la signature de données reçues avant d'être écrites sur la carte. Le noeud *S* (*Sélectionné*) dans une applet, est l'état du programme qui accepte les commandes. Pendant cette étape, l'état vérifie la valeur du byte d'instruction du message reçu pour déterminer quelle méthode doit être appelée. Ceci est représenté par le test sur la transition de l'état *S* à l'état *1*. Ensuite, la machine à état vérifie si le champ *PW1* (*Password 1*) est valide. Si l'instruction de vérification *verifyPW1()* échoue, alors la machine à état entre dans l'état d'erreur *ERROR*. Sinon, le programme effectue la signature. Enfin, la machine à état retrouve l'état de sélection pour attendre une nouvelle commande. Pour illustrer le *missing-check*, supposons que la condition de vérification du password *verifyPW1()* entre le noeud *2* et le noeud *3* n'existe pas. Dans ce cas, un attaquant pourrait pour faire entrer la machine à état dans le noeud *3* avec un *PW1* invalide. C'est précisément ce type de vulnérabilités, les *missing-checks*, que nous cherchons à détecter dans les applets.

Fig. 1: Machine à état partielle pour la commande *COMPUTE_DIGITAL_SIGNATURE*. Il y a des conditions pour passer d'un état à un autre. Les états sont représentés par les noeuds, et les transitions par les liens. Le noeud *S* est l'état *Sélectionné*. Schéma basé sur la spécification d'*OpenPGP*[3]



2.2 État de l'art

Cet état de l'art présente différentes manières d'analyser un code source pour se prémunir de mauvaises implémentations de spécification. Par exemple, certains

travaux se basent sur des méthodes formelles. D'autres se basent du machine learning ou encore sur une analyse statique du code source.

Protection dynamique et hybride Les protections hybrides et dynamiques consistent à filtrer les données controllées par l'utilisateur. Nous avons trouvé ce type de méthode dans le domaine d'application web. Des techniques de protection contre l'attaque *XSS* (*UrlEncoder* ou *HtmlEncoder*[4] par exemple) permettent d'assainir des données reçues. Elles permettent de s'assurer que le format du message et son domaine d'application sont respectés. Un inconvénient à cette technique est qu'il est possible d'oublier des cas à filtrer si une nouvelle attaque apparait. Cette méthode nécessite donc des mises à jour pour plus de précision. De plus, si un attaquant parvenait à obtenir le code source de cette méthode de filtrage, alors il pourrait adapter ses messages pour contourner ce filtre sécurisé. Dans sa thèse[5], Kamel propose d'implémenter une librairie de filtrage de message *API JCXSSFilter* et de l'adapter pour l'application open-source web *ESAPI*[6] de l'*OWASP*. L'utilisation de ce type de librairie ajoute un surcoût d'exécution.

Méthodes formelles Les outils *Z*[7] et *VDM*[8] se basent sur les méthodes formelles. Ils visent à spécifier mathématiquement le comportement attendu d'une séquence d'un système en utilisant des ensembles, des relations et des fonctions. Burde *et al.*[9] présente une expérience sur la validation formelle d'applets *Java Card*. Pour décrire un comportement, l'utilisateur doit annoter ses classes *Java* avec le *Java Modeling Language*. Un inconvénient de cette méthode est qu'il est nécessaire de spécifier le comportement attendu pour toutes les classes. Plus le projet est conséquent, plus cette étape peut être compliquée. De plus, si la spécification change, le développeur doit tout adapter à nouveau pour accorder le code à cette nouvelle spécification.

Analyse statique Une analyse concolique effectue à la fois une analyse concrète et une analyse symbolique pour un code source donné. C'est le cas de *JDart*[10]. Son analyse symbolique découvre les chemins qu'un programme peut suivre, tandis que l'analyse concrète propose des messages valides.
Dans le domaine de l'analyse statique, le suivi de teintes vise à suivre l'évolution d'une donnée d'entrée le long d'un programme. Pour illustrer cette méthode, supposons qu'un utilisateur veuille suivre l'évolution d'une variable. Pour ce faire, il doit la teinter d'une couleur initiale. Ensuite, chaque variable la manipulant reçoit la même coloration. On peut ainsi suivre la propagation de la variable en suivant sa couleur. C'est ce que réalise l'outil *Pixy*. Il permet de suivre des teintes dans les fonctionnalités *PHP* les plus dynamiques. Un inconvénient de cette méthode est que s'il y a beaucoup de chemins dans le flot de contrôle, l'analyse peut ne jamais se terminer.

Fouille de textes La fouille de textes est une technique qui consite à extraire des termes (mots) présents des fichiers, ainsi que leur fréquence. Ensuite, ces fréquences de termes sont corrélées afin de construire un modèle de prédiction

concernant la vulnérabilité d'un des fichiers. Cette méthode est implémentée dans l'outil de Scandariato *et al.*[11].

Machine-learning L'outil *Chucky-ng*[2], basé sur *Chucky*[1] se base sur une technique de machine-learning pour découvrir des vulnérabilités dans un code source. Il extrait et compare les fonctions afin de déterminer par un algorithme non-supervisé celles qui sont vulnérables. *Chucky-ng* se base sur l'algorithme *k-Nearest-Neighbors*[12]. Un avantage important de cet outil est qu'il ne requiert pas de calibration ou de phase d'entraintement avant son utilisation.

2.3 Notre contribution

Il nous semble qu'aucun outil de protection d'applets contre les attaques par fuzzing est disponible au public. Nous ne voulons pas créer un fuzzer pour une raison. Le temps requis pour envoyer une commande via un terminal à la carte est couteux. Nous voulons que notre analyse s'effectue dans les plus brefs délais. Nous avons créé *ChuckyJava*. Ce dernier est une adaptation de *Chucky-ng* pour *Java* pour détecter les *missing-checks* dans les codes sources.

3 Description de *Chucky*

Chucky prend en paramètre trois arguments: le dossier des fichiers sources à analyser, un paramètre k ainsi qu'un objet à observer. Ce paramètre k est utilisé dans l'étape de sélection des voisins. Le dernier paramètre est l'objet que nous voulons analyser. Il peut être une variable, un paramètre, un appel de fonction ou une fonction. S'il s'agit d'une fonction, alors l'analyse est découpée en variables, paramètres et appels de fonctions présents dans cette fonction. *Chucky* retourne alors à l'analyste un score d'anomalie pour les fonctions contenant l'objet à analyser. Pour ce faire, l'algorithme de *Chucky* s'exécute en quatre étapes:

1. Parsing
2. Sélection des voisins
3. Réduction des dimensions
4. Publication du score d'anomalie.

3.1 Parsing

Pendant cette étape, un outil nommé *Joern* parse les fichiers *Java* du dossier passé en argument. *Joern* et *Chucky* sont indépendants. Il crée un *Code Property Graph* qui est l'union de trois graphes distincts: l'arbre de syntaxe abstraite, le graphe de flot de contrôle et le graphe de flot de données. Ce graphe est utilisé pendant l'étape de suivi de teintes.

3.2 Sélection des voisins

Pendant cette étape, *Chucky* regroupe les fonctions (voisins) les plus similaires à celle analysée. L'outil filtre et garde les voisins contenant l'objet sous observation dans un ensemble. Par exemple, un objet souvent utilisé dans les applets *Java Card* est l'Application Protocol Data Unit ($APDU$). Si un analyste souhaite évaluer le comportement des fonctions le manipulant, *Chucky* ne garde de toutes les fonctions que celles utilisant au moins un paramètre $APDU$. Ensuite, la similarité est basée sur la comparaison de symboles d'API. Ces derniers sont des vecteurs représentant les fonctions de l'ensemble avec pour dimension les types des variables, types de paramètres et types d'objets. *Chucky* utilise le *Inverse Document Frequency* pour les valeurs de chaque dimension dans ces vecteurs. De ce fait, les termes plus rares seront plus significatifs que ceux présents dans la majorité des fonctions. En effet, des fonctions utilisant les mêmes symboles d'API rares sont plus à même d'être similaires. Afin de définir si deux fonctions sont similaires, *Chucky* utilise la distance cosinus comme métrique (1), x et y étant des vecteurs. Elle permet de prendre en compte à la fois la distance Euclidienne des vecteurs, mais également leur orientation.

$$cos(\boldsymbol{x}, \boldsymbol{y}) = 1 - \frac{\boldsymbol{x} \cdot \boldsymbol{y}}{||\boldsymbol{x}|| \cdot ||\boldsymbol{y}||}. \tag{1}$$

A présent, *Chucky* a l'ensemble de fonctions similaires par rapport à celle à analyser. Il se sépare donc de ces vecteurs qui lui ont permis d'établir cet ensemble.

3.3 Réduction des dimensions

Cette étape vise à réduire les dimensions des nouveaux vecteurs. Ils sont basés sur les expressions utilisées. Pour cela, *Chucky* va supprimer des fonctions toutes les structures de contrôle n'utilisant pas l'objet observé. Pour y parvenir, il utilise le *Code Property Graph* et va suivre l'évolution de l'object observé au sein de la fonction.
Chucky va normaliser certaines parties de code afin de réduire les faibles différences syntaxiques. Par exemple, les opérateurs de relation binaire ($\leqslant$, $<$, $\geqslant$, $>$) sont maintenant remplacés par \$CMP et les nombres sont normalisés par \$NUM. Une normalisation s'applique aussi pour des arguments de fonction appelées et les valeurs de retour des fonctions. Par exemple: $if(ident <= 1)$ devient $if(ident \quad \$CMP \quad \$NUM)$

3.4 Détection d'anomalies

Pendant cette étape, *Chucky* crée un modèle de normalité. Ce nouveau vecteur a pour dimension le nombre d'expressions normalisées et restantes. Pour chacune de ces dimensions, la valeur est la moyenne des indicateurs de présence ou non de l'expression dans les fonctions de l'ensemble comme le montre l'équation (3). E est l'ensemble des expressions restantes et X est l'ensemble des fonctions voisines. $\varphi(x)$ est la fonction de mapping qui transofmre les voisins dans un

espace vectoriel. La fonction indicateur $I(x, e)$ équivaut à 1 si le voisin x contient l'expression e. Sinon I vaut 0.

$$\varphi : X \to \mathbb{R}^{|E|}, \varphi(x) \mapsto (I(x, e))_{e \in E}. \tag{2}$$

μ est le vecteur de normalité et N l'ensemble des voisins.

$$\mu = \frac{1}{|N|} \sum_{n \in N} \varphi(n), \mu \in \mathbb{R}^{|E|}. \tag{3}$$

La valeur de l'indicateur vaut 1 si l'expression existe dans la fonction. Elle vaut 0 si l'expression n'existe pas. *Chucky* crée ensuite un vecteur de distance d en (4) qui correspond à la valeur de chaque dimension du vecteur de normalité, moins celle correspondant à notre fonction.

$$d = \mu - \varphi(x), d \in \mathbb{R}^{|E|}. \tag{4}$$

Le vecteur de distance peut être vu comme la liste des scores d'anomalie pour les expressions restantes ou normalisées. Pour finir, le score d'anomalie pour la fonction analysée correspond à la valeur maximale de ce vecteur de distance (5). L'intervalle pour ce score est compris entre -1.00 et 1.00. Le tableau 1 précise comment interpréter le score d'anomalie.

$$Score = max(d). \tag{5}$$

Score d'anomalie	Signification
1.00	Notre fonction oublie une expression qui est réalisée chez toutes ses voisines
-1.00	Notre fonction effectue une opération qui n'est faite chez aucune de ses voisines

Tableau 1: Interprétation du score d'anomalie

En pratique, même si nous pourrions nous baser sur le score d'anomalie final d'une fonction, il est préférable de regarder ce score pour chaque expression. En effet, une expression avec un score d'anomalie à 1.00 peut en cacher une autre avec le même score d'anomalie.

4 Adaptation et modification

4.1 Adaptation de *Chucky-ng* en *Java*

Java partage des notions similaires avec celles utilisées en *C++*. Ils partagent par exemple les classes abstraites ainsi que les méthodes virtuelles. Alors que nous retrouvons des notions, nous devons les adapter pour que *Chucky-ng* les parse correctement. Pour ce faire, nous modifions à la fois *Joern* ainsi que *Chucky-ng*. Pour le reste du papier, le nouvel outil créé s'appelle *ChuckyJava*.

Méthodes virtuelles Si une classe hérite de méthodes définies dans l'*API Java* à laquelle nous n'avons pas accès, alors il ne nous est pas possible de les parser. Nous avons créé un outil qui vérifie les classes déclarées et celles effectivement utilisées, puis informe l'utilisateur s'il venait à en manquer.

Code natif En *Java*, un développeur peut écrire du code natif (en *C*) en utilisant la *Java Native Interface*. Nous ne prenons pas en compte dans notre analyse les portions de code en *C* puisque *Java Card* ne permet pas d'utiliser cette interface.

Nouvelles expressions Nous avons eu à prendre en compte de nouvelles expressions. Par exemple, en *C++* la clause *try/catch* n'a pas de bloc *finally*. Cette dernière est exécutée même si le *catch* est appelé. C'est pourquoi nous le traitons comme un bloc d'expression normal, sans contraintes de flot de contrôle. De plus, nous avons eu à ajouter la prise en compte de boucles *for* qui peuvent prendre la forme: *for(Element e: elements)* { *[...]* } et autres.

Nouveaux opérateurs Nous avons implémenté two opérateurs binaires qui existent en *Java* mais pas en *C/C++*.

- Opérateur de comparaison de structure ===
- Opérateur de décalage de bit (bit de signe inclus) >>>

4.2 Modifications de *ChuckyJava*

Modification de l'algorithme Nous apportons une modification sur l'algorithme de sélection des voisins de *ChuckyJava*. Durant l'étape de la sélection des méthodes similaires, nous prenons en compte le type des objets *"castés"* ainsi que le type d'un objet si celui-ci est créé dans un appel de méthode. Par exemple les expressions:

- *int k = myMethod(new OwnerPin(0x03,0x04));*
  le type *OwnerPin* est maintenant pris en compte en *API symbol*.
- *byte c = (byte) 0x80;*
  le type *byte* est pris en compte en *API symbol*.

Switch/cases Nous modifions *ChuckyJava* pour qu'il puisse analyser les *cases* des *switches/cases*. L'outil nous signale à présent s'il venait à manquer un test dans une des applets par rapport aux autres. En effet, l'usage des *switch/cases* est souvent utilisé dans la méthode *process* des applets. C'est une des méthodes appelées lors de la réception d'un message. L'utilisation du *switch/case* permet de tester d'appeler une fonction spécifique en testant l'octet d'instruction du message.

5 Resultats

Notre jeu de données est composé de cinq applets *Java Card* implémentant toutes *OpenPGP* [3]. Parmis ces applets, il y a deux versions d'*OpenPGP* différentes: la

2.0.1 et la 3.3.1. Le nom des applets et leur versions sont exposés dans la tableau 2. Dans le cadre de cette expérience, puisque nous travaillons sur des applets, nous retrouverons au moins la méthode *process* dans toutes les applets. De plus, bien que les structures peuvent varier, on suppose que pour chaque commande, une méthode correspondante sera appelée. Pour chaque méthode, il y en aurait donc 4 autres succeptibles de lui être similaire. C'est pour cette raison que nous choisissons de fixer le nombre de voisins k à 4.

Nom de l'applet	Version d'*OpenPGP*
FluffyPGP	2.0.1
JCOpenPGP	2.0.1
MyPGPid	2.0.1
OpenPGPCard	2.0.1
SmartPGP	3.3.1

Tableau 2: Applets OpenPGP et leur implémentation

Il faut utiliser un objet *APDU* afin de communiquer depuis un terminal avec la *Java Card*. Cet *APDU* contient les informations de communication. Nous en présentons trois octets d'en-tête: le CLA, l'INS et le P1. L'octet CLA (Classe) est utilisé pour définir la classe d'instruction. L'octet INS (Instruction) précise l'instruction à exécuter et enfin l'octet P1 (Paramètre 1) est le premier paramètre pour une instruction. Maintenant, nous analysons les résultats de *ChuckyJava*.

5.1 Une vérification inutile

Le listing 1.1 présente notre résultat avec *ChuckyJava* pour le constructeur *PGPKey* du fichier *PGPKey.java*.

Listing 1.1: Le score d'anomalie, l'expression, la fonction et le fichier concernés

```
-1.00    "( null $CMP $RET )" PGPKey PGPKey.java:38
```

Le score d'anomalie est évalué à -1.00. Cela signifie que "$CMP" (comparaison) est effectuée dans notre fonction mais pas dans les autres. Nous pouvons vérifier ceci avec la portion de code de l'appendice A. A ce niveau dans l'applet, *tmpBuf* n'a pas pu être initialisé, traduisant une vérification inutile de son initialisation. *ChuckyJava* détecte donc un problème d'optimisation.

5.2 Détection de code mort

Nous sommes capables de détecter du code mort avec *ChuckyJava*. Le listing 1.2 montre l'output pour l'analyse de la méthode *process* de l'applet *MyPGPid*.

Listing 1.2: Sortie de *ChuckyJava* pour *MyPGPid*

```
-1.00    "case ISO7816.INS_SELECT"          process MyPGPid.java
```

ChuckyJava retourne un score d'anomalie de -1.00. Cela signifie que la fonction *process* effectue un *case ISO7816.INS_SELECT* qui n'est effectué dans celle des autres. En analysant le code en appendice B, on s'aperçoit qu'un test pour *selectingApplet()* est effectué au début de la méthode. Ce test et *case ISO7816.OFFSET_INS* "retournent" si la valeur de l'INS équivaut à *0xA4*. C'est pourquoi le code de ce *case* (en l'occurrence l'instruction *return*) ne peut pas être atteint.

5.3 Mauvaise utilisation de l'octet CLA

En analysant la sortie de *ChuckyJava* pour la méthode *process*, nous sommes en mesure de déceler une mauvaise utilisation de l'octet CLA. En effet, la spécification d'*OpenPGP*[3] précise que cet octet doit être vérifié avant utilisation. Dans l'applet *MyPGPid*, l'octet est testé avec l'opérateur bit à bit *&* et la valeur 0xFC suite à une assignation.

Listing 1.3: Mauvaise utilisation de l'octet CLA

```
-1.00    "buffer[ISO7816.OFFSET_CLA] = (byte)(buffer[ISO7816.OFFSET_CLA]
    ↪ & (byte) 0xFC)   process MyPGPid.java:347
```

Après analyse du code source, nous avons découvert que cette assignation est effectuée mais au final la valeur de *buffer[ISO7816.OFFSET_CLA]* n'est pas vérifiée. Cette valeur est vérifiée une seule fois dans une méthode. Cela traduit que presque n'importe quelle valeur pour cet octet est possible, bien que la spécification précise que seulement des valeurs (souvent *0x00*, *0x0C*, *0x10* ou encore *0x1C*) peuvent être utilisées. Un attaquant peut être capable de faire entrer le programme dans un état via une transition non prévue par la spécification d'*OpenPGP*.

5.4 Vérification manquante pour l'octet P1

Nous avons découvert une anomalie sur l'octet P1 lors de notre analyse de la fonction *verify* pour notre ensemble d'applets. La sortie de *ChuckyJava* du listing 1.4 nous précise qu'une anomalie existe dans la fonction *verify* du fichier *OpenPGPApplet.java*. La portion de code précise que l'octet n'est pas vérifié dans notre code alors que cela est fait dans les autres méthodes *verify* de l'ensemble.

Listing 1.4: Vérification manquant pour l'octet P1

```
0.67    "buffer[ISO7816.OFFSET_P1] $CMP (byte)($NUM)"    verify
    ↪ OpenPGPApplet.java:413
```

Lors de notre lecture de la spécification pour cette fonction *verify*, nous nous sommes aperçus que l'octet P1 doit avoir *0x00* pour valeur. Il s'agit donc ici

d'une vérification manquante puisque cette vérification est effectuée dans les autres fonctions *verify* de l'ensemble. Nous pouvons retrouver le code de la fonction dans l'appendice C

6 Limitations

6.1 Nombre missing-checks

Pendant la quatrième étape, *ChuckyJava* crée des vecteurs qui contiennent des informations sur la présence ou non d'une expression normalisée. Par exemple, la condition *if (k == 3)* est normalisé par *(k $CMP $NUM)*. S'il y a de multiples fois la variable k comparée avec un nombre, alors l'outil les normalise avec la même expression. Ensuite, la valeur pour la dimension *(k $CMP $NUM)* est égale à 1, peu importe le nombre de comparaison. Nous pouvons voir dans le Listing 1.5 que deux vérifications sont effectuées. Toutefois, seulement une seule est effectuée dans le Listing 1.6. Après l'étape de normalisation, les deux vecteurs représentant respectivement ces deux portions de code contiennent la dimension *(k $CMP $NUM)* avec pour valeur 1. Cela mène donc à la non-détection du *missing-check* du Listing 1.6.

Listing 1.5: Deux comparaisons de nombre avec k

```
public void test(int k)
{
  if(k <= 1)
        callee(1);
  else if(k <= 2)
        callee(2);
}
```

Listing 1.6: Code vulnérable

```
public void test(int k)
{
  if(k <= 1)
        callee(1);
  //Missing check...
  callee(2);
}
```

6.2 Manque de distinction entre variables et constantes

Nous observons qu'il n'est pas possible pour *ChuckyJava* de distinguer une variable d'une constante. Le Listing 1.7 présente un appel à une méthode si le test est vrai. Dans le Listing 1.8, le même appel de fonction utilise une constante plutôt qu'une variable. Parce que c'est une constante, elle a déjà une valeur connue qui lui est affectée. Il n'y a donc pas besoin de la tester. Toutefois, si les autres applets utilisent une variable comme dans le Listing 1.7, alors *ChuckyJava* détecte un *missing-check* dans le code du Listing 1.8. Il s'agit là d'un faux positif.

Listing 1.7: Variable locale

```
if(my_variable > 0)
        myInitMethod(my_variable);
```

Listing 1.8: Constante

```
private static final int MY_CONSTANT = 10;
myInitMethod(MY_CONSTANT);
```

6.3 Noms d'identificateurs

ChuckyJava n'est pas capable de faire la différence entre les identificateurs. Par exemple, nous voulons que *ChuckyJava* analyse les identificateurs portant le nom de *buffer*. Dans ce cas, tous les identificateurs qui ont le même sens (*tmpBuff*, *buf*, etc.) viennent perturber le résultat de l'analyse. Et ce, même si l'utilisation qui en est faite est identique. Une bonne pratique serait de normaliser les identificateurs qui ont la même utilisation y compris pour les noms des méthodes. Il s'agit là de notre priorité afin de réduire le nombre de faux positifs de *ChuckyJava*.

6.4 Problème de structure

Prenons l'exemple du Listing 1.9 et du Listing 1.10. Ces deux portions de code sont sémantiquement identiques. Toutefois, dans le premier, l'appel à la méthode *apdu.getBuffer()* est effectué en dehors de la méthode *ma_methode(APDU apdu)* puisque *buffer* est déclaré en variable globale. Dans le second, cet appel est fait directement dans la méthode *ma_methode(APDU apdu)*. Dans un ensemble d'applets à analyser, on cherche à comparer les méthodes *ma_methode*. Dans cet ensemble, les applets sont réalisées pour faire l'appel à *apdu.getBuffer()* une unique fois, comme dans le Listing 1.9. Toutefois, dans ce même ensemble, une seule applet fait comme dans le Listing 1.10. Dans ce cas, *ChuckyJava* va assigner un score d'anomalie à 1.00 pour la méthode *ma_methode* de l'applet réalisée comme dans le Listing 1.10. En effet, elle est la seule à faire appel à *apdu.getBuffer()*. Dans le cas inverse, si des applets de type Listing 1.10 sont en majorité dans notre ensemble avec une seule applet de type Listing 1.9, alors *ChuckyJava* va également assigner un score d'anomalie de 1.00 à la méthode *ma_methode* du Listing 1.9. Dans les deux cas, il s'agit d'un faux positif. Il s'agit d'une limitation concernant la structure d'une applet.

Listing 1.9: *apdu.getBuffer()* en dehors de *ma_methode*

```
private byte[] buffer = null;
protected void process(APDU apdu){
        [...]
        buffer = apdu.getBuffer();
        switch(buffer[ISO7816.OFFSET_INS]){
                case INS_VERIFY:
                        ma_methode(apdu);
```

```
                        break;
        [...]
        }
}
private void ma_methode(APDU apdu)
{
        if(buffer[ISO7816.OFFSET_P1] == (byte) 0x20)
                operation01(apdu);

}
```

Listing 1.10: *apdu.getBuffer()* dans *ma_methode*

```
protected void process(APDU apdu){
        [...]
        byte[] buffer = apdu.getBuffer();
        switch(buffer[ISO7816.OFFSET_INS]){
                case INS_VERIFY:
                        ma_methode(apdu);
                        break;
        [...]
        }
}
private void ma_methode(APDU apdu)
{
        byte[] buffer = apdu.getBuffer();
        if(buffer[ISO7816.OFFSET_P1] == (byte) 0x20)
                operation01(apdu);

}
```

7 Conclusion

ChuckyJava a des résultats promettant pour détecter des vulnérabilités. Toutefois, l'outil a quatre limites. Pour résoudre la première limite, nous aurons à changer la formule du score d'anomalie. Nous devons prendre en compte le nombre de tests présents dans une fonction. Pour résoudre la seconde limite, il faudrait prendre en compte la nature de l'identificateur manipulé. Il faut que *ChuckyJava* distingue s'il s'agit d'une constante ou d'une variable. Nous sommes actuellement en train de normaliser les noms identificateurs similaires entre eux afin de limiter les faux positifs retournés par *ChuckyJava*. Pour ce faire, nous regardons diverses techniques issues de la détection de plagiat ou de la détection de portions de codes similaires via des techniques de fouille de textes. D'autres techniques se basent sur la détections de séquences similaires d'*ADN* en utilisant des arbres de suffixes. Concernant la dernière limite, il faudra prendre en compte le graphe d'appel pour vérifier les appels effectués au préalable pour chaque méthode analysée.

References

1. F. Yamaguchi, C. Wressnegger, H. Gascon, and K. Rieck, "Chucky: Exposing missing checks in source code for vulnerability discovery,"

2. F. Zhao, K. Yang, and J. Ren, "Improving neighbourhood quality for chucky: an empirical study,"
3. A. Pietig, *Functional Specification of the OpenPGP application on ISO Smart Card Operating Systems*.
4. OWASP, "Xss (cross site scripting) prevention cheat sheet." https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet.
5. N. Kamel, "Sécurité des cartes à puce à serveur web embarqué," *PhD thesis, Université of Limoges*, 2012.
6. "Owasp enterprise security api," 2009.
7. J. Spivey, "Understanding z: a specification language and its semantics, vol. 3. cambridge university press 1988,"
8. C. Jones, "Systematic software development using vdm," *vol. 2. Orentice-Hall Englewood Cliffs, NJ*, 1986.
9. L. Burdy, A. Requet, and J. Lanet, "Java applet correctness: A developer-oriented approach," *International Symposium of Formal Methodes Europe*, 2003.
10. K. Luckow, M. Dimjasevic, D. Giannakopoulou, F. Howar, M. Isberner, T. Kahsai, Z. Rakamaric, and V. Raman, "Jdart: A dynamic symbolic analysis framework,"
11. R. Scandariato, J. Walden, H. A., and J. W., "Predicting vulnerable software components via text mining,"
12. S. Sayad, "K nearest neighbors." http://chem-eng.utoronto.ca/ datamining/Presentations/KNN.pdf.

A Constructeur de *PGPKey*

```
private static byte[] tmpBuf;
[...]
public PGPKey() {
  key = new KeyPair(KeyPair.ALG_RSA_CRT, KEY_SIZE);
  fp = new byte[FP_SIZE];
  Util.arrayFillNonAtomic(fp, (short) 0, (short) fp.length, (byte) 0);
  Util.setShort(attributes, (short) 1, KEY_SIZE);
  Util.setShort(attributes, (short) 3, EXPONENT_SIZE);
  //The useless check
  if(tmpBuf == null) {
   tmpBuf = JCSystem.makeTransientByteArray((short) (KEY_SIZE_BYTES / 2)
       ↪ , JCSystem.CLEAR_ON_DESELECT);
  }
 }
```

B Fonction *process* de l'applet *MyPGPid*

```
public void process(APDU apdu) {
    byte[] buffer = apdu.getBuffer();
    short lc;
    boolean status = false;

    // ignore the applet select command dispached to the process
    if (selectingApplet()) {
        return;
    }

    buffer[ISO7816.OFFSET_CLA] = (byte)(buffer[ISO7816.OFFSET_CLA] & (
        ↪ byte)0xFC);
```

```
if (buffer[ISO7816.OFFSET_INS] == GET_RESPONSE) {
    if (remainingDataLength <= 0) {
        ISOException.throwIt(ISO7816.SW_CONDITIONS_NOT_SATISFIED);
    }
    else { sendData(apdu, tmpData, remainingDataLength);}
    return;
} else {
    remainingDataLength = 0;
    remainingDataOffset = 0;
}

switch (buffer[ISO7816.OFFSET_INS]) {
    case ISO7816.INS_SELECT:
        return;
    case GET_DATA:
       getData(apdu);
       return;
    case PUT_DATA:
        putData(apdu);
        return;
    case PUT_DATA_CHAINING:
        putDataChaining(apdu);
        return;
    case VERIFY:
        if (buffer[ISO7816.OFFSET_P1] != 0) { ISOException.throwIt(
            ↪ ISO7816.SW_WRONG_P1P2); }
        lc = apdu.setIncomingAndReceive();
        if (lc == 0) {
            ISOException.throwIt(ISO7816.
                ↪ SW_SECURITY_STATUS_NOT_SATISFIED);
        }
        switch (buffer[ISO7816.OFFSET_P2]) {
            case (byte)0x81:
                if (chv1.getTriesRemaining() == (byte)0) {
                    ISOException.throwIt(SW_PIN_BLOCKED);
                }
                status = chv1.check(buffer, (short)ISO7816.
                    ↪ OFFSET_CDATA, (byte)lc);
                break;
            case (byte)0x82:
                if (chv2.getTriesRemaining() == (byte)0) {
                    ISOException.throwIt(SW_PIN_BLOCKED);
                }
                status = chv2.check(buffer, (short)ISO7816.
                    ↪ OFFSET_CDATA, (byte)lc);
                break;
            case (byte)0x83:
                if (chv3.getTriesRemaining() == (byte)0) {
                    ISOException.throwIt(SW_PIN_BLOCKED);
                }
                status = chv3.check(buffer, (short)ISO7816.
                    ↪ OFFSET_CDATA, (byte)lc);
                break;
            default:
                ISOException.throwIt(ISO7816.SW_WRONG_P1P2);
        }
        if (!status) { ISOException.throwIt(ISO7816.
            ↪ SW_SECURITY_STATUS_NOT_SATISFIED);}
        return;
    case GENERATE_ASYMMETRIC_KEY_PAIR:
        generateAssymetricKeyPair(apdu);
        return;
    case PERFORM_SECURITY_OPERATION:
        performSecurityOperation(apdu);
        return;
    case CHANGE_REFERENCE_DATA:
        /* Fall through */
```

```
        case RESET_RETRY_COUNTER:
            changeResetChv(apdu);
            return;
        case INTERNAL_AUTHENTICATE:
            if (buffer[ISO7816.OFFSET_P1] != 0 || buffer[ISO7816.
                ↪ OFFSET_P2]
                    != 0) {
                ISOException.throwIt(ISO7816.SW_WRONG_P1P2);
            }
            if (!chv2.isValidated()) {
                ISOException.throwIt(ISO7816.
                    ↪ SW_SECURITY_STATUS_NOT_SATISFIED);
            }
            lc = receiveData(apdu, tmpData);
            sig.init(keyAuth.getPrivate(), Cipher.MODE_ENCRYPT);
            lc = sig.doFinal(tmpData, (short)0, lc, tmpData, (short)0);
            sendData(apdu, tmpData, lc);
            return;
        case GET_CHALLENGE:
            if (buffer[ISO7816.OFFSET_P1] != 0 || buffer[ISO7816.
                ↪ OFFSET_P2]
                    != 0) {
                ISOException.throwIt(ISO7816.SW_WRONG_P1P2);
            }
            lc = apdu.setOutgoing();
            random.generateData(tmpData, (short) 0, lc);
            apdu.setOutgoingLength(lc);
            apdu.sendBytesLong(tmpData, (short) 0, lc);
            return;
        //case EXPORT_KEY_PAIR:
            //exportKeyPair(apdu);
            //return;

        case INS_CARD_READ_POLICY:
            ReadPolicy(apdu);
            return;
        case INS_CARD_KEY_PUSH:
            KeyPush(apdu);
            return;

        default:
            ISOException.throwIt(ISO7816.SW_INS_NOT_SUPPORTED);
    }
}
```

C Fonction *verify* d'*OpenPGPApplet*

```
private void verify(APDU apdu, byte mode) {
 if (mode == (byte) 0x81 || mode == (byte) 0x82) {
  // Check length of input
  if (in_received < PW1_MIN_LENGTH || in_received > PW1_MAX_LENGTH)
   ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);

  // Check given PW1 and set requested mode if verified succesfully
  if (pw1.check(buffer, _0, (byte) in_received)) {
   if (mode == (byte) 0x81)
    pw1_modes[PW1_MODE_NO81] = true;
   else
    pw1_modes[PW1_MODE_NO82] = true;
  } else {
   ISOException
   .throwIt((short) (0x63C0 | pw1.getTriesRemaining()));
  }
 } else if (mode == (byte) 0x83) {
```

```
    // Check length of input
    if (in_received < PW3_MIN_LENGTH || in_received > PW3_MAX_LENGTH)
      ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);

    // Check PW3
    if (!pw3.check(buffer, _0, (byte) in_received)) {
      ISOException
      .throwIt((short) (0x63C0 | pw3.getTriesRemaining()));
    }
  } else {
    ISOException.throwIt(ISO7816.SW_INCORRECT_P1P2);
  }
}
```