Projet universitaire

Auteurs :

- BERTRAND Pierre-Louis
- NEONAKIS Ionas
- QUETIER Thomas
- TOULARHMINE Samir

Vérificateur de bytecode

19 Janvier 2022

Vue d'ensemble

Ce projet avait pour but de développer un vérificateur de bytecode Android en Python. Cette vérification devait s'effectuer sur une un classe choisie par l'utilisateur, au sein d'une APK Android.

Le vérificateur doit permettre plusieurs analyses :

- l'analyse 1 permet de vérifier qu'à tout moment les instructions exécutées sont cohérentes avec le type des valeurs contenues dans les registres,

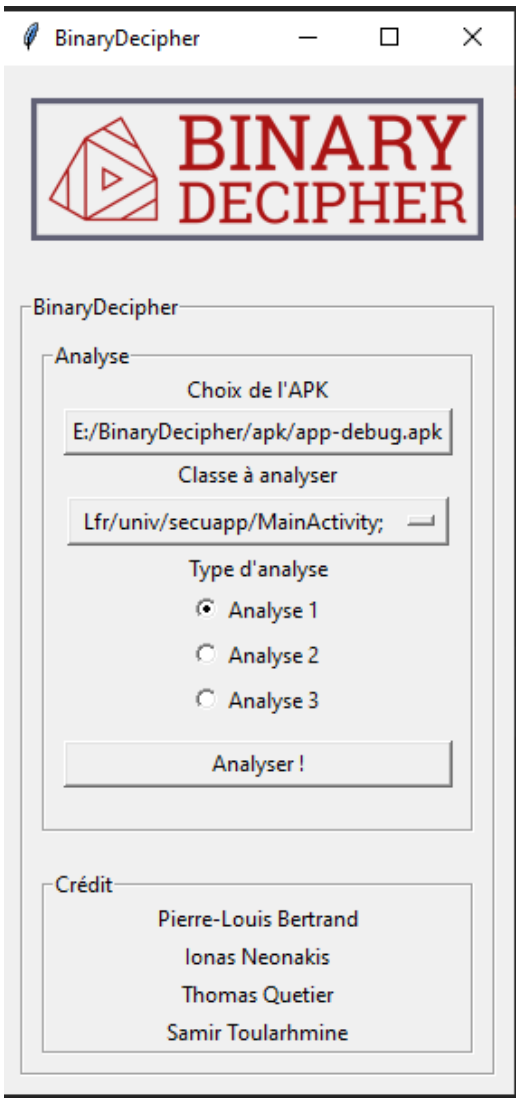
- l'analyse 2 permet de vérifier qu'il n'y a aucun accès à un objet alloué mais non initialisé,

- l'analyse 3 permet de mettre en évidence les communications de l'application (permissions, sollicitations internes ou externes).

Présentation de notre solution

Pour répondre à cette problématique nous avons conçu une interface graphique pour rendre le programme accessible. Réalisée avec Tkinter, une librairie de python, notre application est facilement exportable sur différents systèmes d'exploitations.

Via celle-ci, l'utilisateur peut parcourir son système de fichier afin de sélectionner l'APK qu'il souhaite analyser, ainsi que la classe à traiter. Après avoir choisi l'une des trois analyses, l'analyse peut commencer.



Interface de l'application

Une fois l'analyse effectuée, le rapport est généré et l'utilisateur peut enchaîner avec une autre analyse s'il le souhaite.

Analyse de l'APK

Pour pouvoir analyser le bytecode, nous commençons par analyser l'APK fourni par l'utilisateur. Pour cela, nous utilisons la librairie python *Androguard,* notamment la méthode *AnalyzeAPK* qui renvoie trois objets :

- *androguard.core.bytecodes.apk.APK* : représentant l'APK
- *androguard.core.bytecodes.dvm.DalvikVMFormat* : représentant le contenu des fichier DEX
- *analysis.Analysis* : Un support d'analyse

C'est ce deuxième objet qui nous permet de récupérer la classe, avec le nom fourni par l'utilisateur.

Une fois dans la classe, nous récupérons les méthodes sous forme d'objet *Androguard*. Cela nous permet de récupérer des informations pertinentes pour les analyses, comme le type de retour ou encore si la classe est statique par exemple. Pour rendre notre analyse plus simple, nous avons créé une architecture spécifique.

Notre architecture

Notre programme repose sur deux classes principales : *Methode.py* et *Instruction.py*.

Une méthode est notamment composée d'une liste d'instructions, mais aussi des registres ainsi qu'un flot correspondant au déroulement de ses instructions.

Les instructions quant à elles sont des objets destinés à recevoir les informations pertinentes pour nos analyses. Elles sont initialisées à partir des objets *Instruction* de la librairie *Androguard*. Chacunes des instructions sont différentes, de par leur sémantique ou leurs attributs. Pour pouvoir en traiter un maximum, nous avons dû les gérer séparément.

```
elif self._name == 'instance-of':
   # instance-of vA, vB, type@CCCC
   # A: destination register (4 bits)
   # B: reference-bearing register (4 bits)
   # C: type index (16 bits)
   self._register.extend([instr.A, instr.B])
   self._type = instr.cm.get_type(instr.CCCC)
   self._string = f"instruction {self._name} stock dans {self._register[0]}
1 si l'élément référencé dans v{self._register[1]} est du type self._type}"
```

Exemple ici avec l'instruction *instance-of*. Sur la documentation du bytecode *Android Dalvik* on apprend que cette instruction possède 3 attributs différents : A, B et C. On sait que les attributs A et B correspondent aux registres utilisés par l'instruction et C contient l'index d'un objet *Type* accessible via le *ClassManager* de l'instruction.

Ces informations nous seront utiles pour les différentes analyses : nous stockons donc dans un attribut *self._register* la liste des registres utilisés, mais aussi le type. Enfin, on décrit le fonctionnement de l'instruction dans une string qui sera affichée lors des analyses pour suivre l'état du programme.

L'une des difficultés majeures du projet, c'est l'instantiation correcte des différentes instructions. Ces dernières n'ayant pas le même nombre d'attributs, ou alors sur des espaces mémoire plus grands (8, 16, 32 ou encore 64 bits), une gestion globale est complexe. En les identifiant par nom, nous sommes parvenus à les traiter en majorité.

Une fois toutes les instructions ajoutées à l'objet Méthode, nous créons le flow du programme.

```
def compute_succ(self):
   offset = 0
   for instr in self._instructions:
       destination = []
       if instr.get_name() == "":  # Gérer les cas ou la destination n'est
pas explicite (comme les if ou les goto)
           pass
           # destination = offset + valeur à déterminer (par exemple le
goto a une destination spécifique
       elif instr.get_name()[:2] == "if":
           destination = [offset + instr.get_destination(), offset +
instr.get_length()]
       elif "goto" in instr.get_name():
           destination.append(instr.get_destination() + offset)
       elif "return" in instr.get_name():
           destination = []
       else:
           destination.append(
               offset + instr.get_length())  # Juste l'instruction d'après
(car c'est une instruction sequentiel
       self._succ[offset] = (instr, destination, offset)
       offset += instr.get_length()  # Mise à jour de l'offset (adresse de
l'instruction suivante
```

Dans la plupart des cas, les instructions s'exécutent dans un ordre séquentiel sauf avec quelques instructions comme sur l'exemple ci-dessus, où les *if* ont plusieurs successeurs. Ici, l'instruction `instr.get_destination()` nous renvoie la valeur du "saut" à effectuer par le *goto* (défini lors de son instanciation dans la classe *Instruction.py*).

Une fois le flow créé, on peut commencer les analyses.

Première analyse - Les registres

Pour l'analyse des registres, nous parcourons les différentes instructions en mettant à jour l'état des registres avec leurs types et leurs valeurs (quand nous pouvons la calculer). Une fois l'instruction traitée, on traite ses successeurs. Si une instruction a plusieurs parents, on merge l'état des registres de ses parents (si possible, sinon on retourne une erreur).

```
elif 'const-string' in curr_instr.get_name():
   self._etat_reg[curr_instr.get_register()[0]] = ('Ljava/lang/String;',
curr_instr.get_string_value())

elif curr_instr.get_name() == 'return-void':
   if not self._informations['return'] == 'void':
       self.set_verbose_error(curr_instr, 'Erreur dans les
registres(méthode :  ' + curr_instr.get_name() + ', le type de retour ne
correspond pas.)')
```

Ici par exemple, on traite l'instruction *const-string* en modifiant la valeur du registre utilisé par l'instruction. *return-void* renvoie une erreur si le type de retour de la méthode en cours n'est pas void.

Une fois fini, le rapport est disponible dans *out/*.

Cependant, certaines erreurs ne sont pas traitées correctement, notamment celle liée à l'héritage. Cette erreur intervient notamment sur le "init"' qui déclenche une erreur de type liée au contexte de l'invoke.

Aussi, certaine instruction ne sont pas gérées : les invoke polymorphic, les filled-new-array/range, les invoke-custom , packed-switch, sparse-switch, les cmp, les iget iput, ainsi que les const method handle et type.

Deuxième analyse - Bonne initialisation des objets

Cette analyse est un premier jet d'une analyse qui pourrait être plus approfondie. Pour réaliser cette analyse (intra-procédurale), nous commençons par itérer sur les instructions de chaque méthode.

À chaque instruction de type "new-instance" ou "new-array", nous insérons dans un dictionnaire une nouvelle clé correspondant au numéro du registre avec en valeur un couple (TypeObjet, True).

Pour chaque instruction de type "move-object" (qui survient après une instruction "new-instance"), nous mettons à jour la nouvelle valeur du registre contenant l'objet initialisé.

Enfin, pour chaque instruction de type "invoke-virtual", nous vérifions si le registre de l'instruction est bien dans notre dictionnaire et si l'objet est bien initialisé en interrogeant la seconde valeur du couple.

Cela nous permet de savoir si une méthode est appelée sur un objet non instancié et cela est vérifiable avec l'APK TEST_APK_1.apk en analysant la classe CheckInheritance et en utilisant l'analyse 2.

Exemple du fichier rapport en sortie :

```
Nom de la méthode : check
Appel de méthode sur un objet non initialisé :
invoke-virtual appelle la méthode :
['Lfr/univ/test_apk_analyse_1_ok/inheritance/Animal;', 'genericSound',
['()', 'V']] et utilise les registres :  v0

Voici les objets qui ont été correctement initialisés :
Registre :  1  de type  Lfr/univ/test_apk_analyse_1_ok/inheritance/Chien;
```

Troisième analyse - Les communications

Lors de cette analyse nous avons utilisé les méthodes fournies par androguard afin de récupérer les permissions qui sont demandées dans l'apk. On peut ainsi afficher toutes les permissions officielles d'Android présentes dans le fichier manifest.xml

De plus nous avons récupéré les permissions déclarées, c'est-à-dire des permissions complémentaires au permissions officielles d'Android qui sont égalements déclarées dans le fichier manifest.xml

Génération des rapports

Lors de l'analyse d'une classe, notre programme génère 2 types de rapport.

Le premier rapport affiche le résumé de chaque méthode, c'est-à-dire :

- le nombre de registres,
- le type de retour,
- une explication sommaire de chaque instruction,
- la description technique des instructions vues par Androguard,

Après ce résumé, le rapport présente le contenu des registres après chaque instruction, sous la forme d'un dictionnaire : la clé correspond au numéro du registre, et la valeur est un tuple. Ce tuple correspond le type de la valeur contenue dans le registre (None si le registre est vide), et contient également sa valeur (None si c'est impossible de la connaître).

Le second rapport contient le détail de chaque erreur rencontrée lors de l'analyse de la classe.

Nous avons décidé de générer les deux rapports lors de chaque analyse, car toutes les classes rencontrent des erreurs lors des méthodes init (car il nous est impossible de tester l'héritage).

D'un point de vue technique, le premier rapport est généré en récupérant la sortie standard du programme, où nous affichons au fur et à mesure de l'analyse les informations. Le second rapport est généré via l'utilisation d'une classe partagée entre les différents fichiers, qui contient une string contenant le détail de chaque erreur.