Rapport Projet de SAN

Master 2 IMIS

HERVOCHON Corentin

RIBARDIÈRE Tom

java -jar apktool.jar b [chemin dossier smali modif] -o [nom_apk_modif].apkWALCAK Ladislas

Table des matières

I. Architecture

Lors de la conception de notre outil, nous avons fait en sorte que chacune des parties soient indépendantes. C'est donc pour cela que notre outil est décomposé en trois parties qui sont, le **main**, l'**engine** et l'**analyser**.

1. Partie 'Main'

Dans cette première partie, plus utilitaire, on se charge de décomposer la ligne de commande et de l'analyser (en utilisant la librairie *argparse*). Ensuite, le résultat de cette décomposition est utilisé lors de l'ouverture de l'APK pour trouver la classe demandée, et construire un dictionnaire de données concernant cette classe mais aussi l'APK en général. Enfin, elle transmet ce dictionnaire de données à l'Engine.

2. Engine

Le travail de cette partie est de sélectionner l'objet 'Analyser' correspondant à l'analyse demandée. Une fois ce choix fait, elle parcourt la totalité des méthodes contenues dans la classe. Pour chacune des méthodes, dépendamment de l'analyse, soit elle construit le flow de la méthode (Analyse 1 & 2), ou bien elle parcourt celle-ci dans l'ordre des définitions (Analyse 3). Pour chacunes de instructions, elle lance l'analyse grâce à l'objet 'Analyser' créé. Le calcul des *in* et des *out* est donc totalement indépendant du moteur d'analyse, et du type de mémoire.

3. Analyser

Enfin, cette dernière partie est composée des différents analyseurs. La classe de base 'Analyser' est une classe abstraite contenant des méthodes et attributs nécessaires à toutes les analyses.

Chaque analyse est implémentée en créant une nouvelle classe implémentant 'Analyser'. Pour chaque analyse, on utilise une instruction '*match*' (disponible avec Python 3.10), afin de rediriger l'analyse de l'instruction vers la bonne méthode. Cette manière de faire permet de simuler un paterne visiteur, impossible en python.

Dépendamment du type d'instruction, l'outil va vérifier un ensemble de propriétés (e.g Le numéro de registre est valide, la valeur est du type attendu, appel de fonction avec les bons types de paramètres, un move-object est seulement appelé sur un type object, …). Dans le cas d'une erreur, le programme va s'arrêter, et afficher un message détaillant l'erreur rencontrée, ainsi qu'un ExitCode correspondant.

Les analyses 1 & 2 étant très semblables, nous avons, par manque de temps, copié-collé le contenu de l'analyse 1. C'est donc pour cela que l'analyse 2 effectue en même temps l'analyse 1. Cela nous paraît cohérent (L'on ne peut pas analyser la bonne initialisation d'un objet sans auparavant vérifier que le registre dans lequel on le sauvegarde est valide).

Pour ce qui est de l'analyse 3, nous n'avons pas implémenté le fait de passer un fichier contenant la liste des classes à analyser. Elle s'utilise donc comme les deux précédentes analyses.

Par manque de temps, encore une fois, il ne nous a pas été possible d'implémenter toutes les instructions possibles. Un tableau contenant l'ensemble des instructions implémentées est disponible dans le README.md.

Pour finir, lors de la fusion des mémoires des prédécesseurs, l'impossibilité d'obtenir la hiérarchie de classe et le fait que l'on ne prend pas en compte les types dégénérés des types primitifs peut poser des problèmes dans certains cas spécifiques.

II. Choix d'implémentation et problèmes

1. Test d'héritage

Comme dit lors de notre échange par mail, il semblerait que l'outil Androguard ne soit pas capable d'extraire la hiérarchie de classe interne à Android ou Java. De ce fait, le test permettant de savoir si une classe est une sous-classe d'une autre est difficilement faisable de manière fiable.

Nous avons donc décidé, par manque de temps, de considérer cette relation comme vraie dans les cas où il nous est impossible d'obtenir la hiérarchie de classe. C'est donc pour ça que, par exemple, notre code considère java.lang.String et java.lang.Integer comme semblable.

Si nous avions eu plus de temps, il aurait fallu mettre l'intégralité de la hiérarchie Java et Android au sein de l'application.

2. Duplication de code

En raison des délais courts, nous avons préféré dupliquer du code qui fonctionne, en pensant à optimiser le contenu plus tard. Cependant, nous n'avons pas eu le temps d'effectuer ce refactoring.

3. Types des constantes

Lors du développement, nous avons trouvé très d'informations quand au types des variables mises dans les registres lors des instructions 'const', 'const/16', …
De ce fait, nous avons, dans la plupart des cas, considéré que ces 'literal value' comme étant des ints.