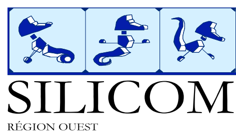


INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

centre de recherche RENNES - BRETAGNE ATLANTIQUE

SGDN

Projet: JAVASEC

Type : rapport d'étude

Rapport d'étude sur les modèles d'exécution de Java

Référence	:	JAVASEC_NTE_003
Version	:	1.2
Nb pages	:	73
Date	:	14 octobre 2009

TABLE DES MATIÈRES

1 Introduction 5
1.1 Objet du document . . . 5
1.2 Présentation du contexte . . . 5
1.3 Organisation du document . . . 6

2 Glossaire, Acronymes 7

3 Présentation des différents modèles d'exécution 8

4 Modèle d'exécution par émulateur 11
4.1 Principes de la compilation . . . 13
4.2 Démarrage de l'application Java, chargement de classes et initialisation . . . 13
4.3 *Bytecode Verifier* . . . 13
4.4 Exécution de *bytecode* . . . 14
4.4.1 Interprétation . . . 14
4.4.2 Compilation « à la volée » (*JIT*) . . . 16
4.4.3 Interprétation et optimisation dynamique du *bytecode* . . . 17
4.4.4 Optimisations réalisées et contraintes sémantiques . . . 17
4.4.4.1 Dépliage de méthode (*inlining*) . . . 18
4.4.4.2 Vérifications dynamiques et gestion des exceptions . . . 19
4.4.4.3 Synchronisation . . . 20
4.4.4.4 Annotations pour l'optimisation . . . 21
4.4.4.5 Optimisations classiques non-spécifiques à Java . . . 21
4.4.5 Frontière statique/dynamique - Problématique de stockage et d'intégrité du code natif . . . 22
4.5 Gestion des *threads* . . . 23
4.5.1 *Threads* OS . . . 23
4.5.2 Implémentation des *threads* Java par la JVM sans utilisation des *threads* OS . . . 23
4.5.3 Implémentation des *threads* Java par la JVM avec utilisation des *threads* OS . . . 24
4.5.4 *Threads* de la JVM . . . 24
4.5.5 Implémentation des mécanismes de synchronisation Java . . . 25
4.5.5.1 Implémentation du mot-clé `synchronized` . . . 25
4.5.5.2 Implémentation des primitives de synchronisation du *package* `java.util.concurrent` . . . 25
4.5.6 Conclusion . . . 26
4.6 Gestion de la mémoire . . . 26
4.6.1 Problématique de l'allocation . . . 26
4.6.1.1 Zones mémoire locales . . . 26
4.6.1.2 Zones mémoire globales . . . 27
4.6.1.3 Implémentation des références . . . 27
4.6.2 Problématique de la libération . . . 28

4.6.2.1 Comptage de références 28
4.6.2.2 Exploration 29
4.6.2.3 Optimisations 32
4.6.3 Conclusion 33
4.7 Intégration avec les mécanismes de l'OS 33

5 Modèle d'exécution natif **35**
5.1 Introduction 35
5.2 Historique 35
5.3 Implémentations 36
5.4 Architecture canonique 37
5.5 Architecture alternative 37
5.6 Problématique de stockage et d'intégrité du code natif 39
5.7 Gestion des *threads* 39
5.8 Gestion de la mémoire 40
5.9 Gestion du chargement de classes 40
5.10 Bibliothèques de support à l'exécution 40
5.11 Implémentation native de la bibliothèque standard 41
5.12 Conclusion 41

6 Modèles d'exécution par processeur spécialisé **43**
6.1 Introduction au modèle d'exécution 43
6.2 Périmètre de l'étude 44
6.3 Solutions étudiées 44
6.4 picoJava 45
6.4.1 Historique 45
6.4.2 Implémentation 45
6.4.3 Version du langage Java supporté 45
6.4.4 Evolution 45
6.5 aJile 46
6.5.1 Historique 46
6.5.2 Implémentation 46
6.5.3 Version du langage Java supporté 47
6.5.4 Evolution 47
6.6 Imsys Technologies AB 48
6.6.1 Historique 48
6.6.2 Implémentation 48
6.6.3 Version du langage Java supporté 49
6.6.4 Evolution 49
6.7 JOP 49
6.7.1 Historique 49
6.7.2 Implémentation 50
6.7.3 Version du langage Java supporté 51
6.7.4 Evolution 51
6.8 Jazelle 51
6.8.1 Historique 51

6.8.2 Implémentation 52
6.8.3 Version du langage Java supporté 53
6.8.4 Evolution 53
6.9 Autres implémentations 54
6.10 Tableau de synthèse 54

7 Modèle d'exécution JavaCard 56
7.1 Un sous ensemble du langage Java 56
7.2 Cycle de vie d'une applet 57
7.3 Dynamique de fonctionnement 57
7.4 Une machine virtuelle adaptée 57
7.5 Mécanismes de sécurité 58

8 Problématique de la décompilation 60
8.1 Décompilation de *bytecode* 61
8.2 Techniques d'offuscation 64
8.2.1 Offuscation de la présentation du *bytecode* 64
8.2.1.1 Renommage 64
8.2.1.2 Renommage avec surcharge 65
8.2.1.3 Suppression des informations de débogage 66
8.2.2 Offuscation des données 66
8.2.2.1 Chiffrement des chaînes littérales 66
8.2.3 Offuscation du flot de contrôle 67
8.2.3.1 Modification des sauts conditionnels 67
8.2.3.2 Mise à plat de la hiérarchie des classes 67
8.2.4 Offuscations/transformations préventives 68
8.3 Analyse des produits d'offuscation 68
8.3.1 Résultats de l'analyse 69
8.3.2 Démarche d'utilisation et limitations 70
8.4 Conclusion sur la décompilation 71

1 INTRODUCTION

1.1 Objet du document

Ce document est réalisé dans le cadre du Projet JAVASEC, relatif au marché 2008.01801.00 .2.12.075.01 notifié le 23/12/2008. Il correspond au livrable technique contractuel émis au titre du poste 1 : rapport sur les modèles d'exécution Java en version finale (identifiant 1.2.2 dans le CCTP).

Il présente un état de l'art des modèles d'exécution Java, notamment pour étudier la préservation des propriétés intrinsèques du langage Java.

1.2 Présentation du contexte

Java est un langage de programmation orienté objet développé par Sun. En plus d'un langage de programmation, Java fournit également une très riche bibliothèque de classes pour tous les domaines d'application de l'informatique, d'Internet aux bases de données relationnelles, des cartes à puces et téléphones portables aux superordinateurs. Java présente des caractéristiques très intéressantes qui en font une plate-forme de développement constituant l'innovation la plus intéressante apparue ces dernières années.

Dans le cadre de ses activités d'expertise, de définition et de mise en œuvre de la stratégie gouvernementale dans le domaine de la sécurité des systèmes d'information, l'ANSSI souhaite bénéficier d'une expertise scientifique et technique sur l'adéquation du langage Java au développement d'applications de sécurité, ainsi que d'une démarche permettant d'améliorer la confiance vis-à-vis de la sécurité de ces applications.

Ce document a pour objectif d'étudier et de comparer les différents modèles d'exécution de Java en termes de sécurité et d'adéquation avec le cadre d'utilisation fixé. Il précise, en les comparant, les différentes stratégies d'implémentation permettant de garantir les propriétés de sécurité identifiées dans le « Rapport d'étude sur le langage Java » [6].

1.3 Organisation du document

Chapitre	Intitulé	Contenu
2	Glossaire, Acronymes	Description des différents acronymes du document.
3	Présentation des différents modèles d'exécution	Présentation de la spécification de la JVM et des différentes stratégies d'implémentation de ce modèle.
4	Modèle d'exécution par émulateur	Présentation des spécificités du mode d'exécution par émulateur (couramment appelé mode d'exécution JVM).
5	Modèle d'exécution natif	Présentation des spécificités du mode d'exécution par exécutable autonome.
6	Processeurs spécialisés	Historique des différents processeurs Java, fonctionnalités marquantes.
7	Modèle d'exécution Javacard	Environnement d'exécution Java pour carte à puce.
8	Problématique de la décompilation	Etude de la problématique de la décompilation de *bytecode*. Analyse des techniques et des outils d'offuscation.

2 GLOSSAIRE, ACRONYMES

Acronyme	Définition
API	Application Programming Interface : interface de programmation
AOT	Compilation Ahead Of Time : compilation en avance de phase
CAP	Converter Applet
CLDC	Connected Limited Device Configuration
FPGA	Field-Programmable Gate Array : circuit intégré logique qui peut être reprogrammé après sa fabrication
JCP	Java Community Process ou Java CertPath (suivant le contexte)
JCVM	JavaCard Virtual Machine
JDK	Java Development Kit : environnement de développement Java
JIT	Compilation Just In Time : compilation à la volée
JNI	Java Native Interface : interface de programmation pour l'intégration des fonctions natives
JRE	Java Runtime Environment : environnement d'exécution Java
JSR	Java Specification Request
JVM	Java Virtual Machine : machine virtuelle Java

3 PRÉSENTATION DES DIFFÉRENTS MODÈLES D'EXÉCUTION

La plate-forme d'exécution Java repose sur trois spécifications :

- la spécification du langage Java [10] ;
- l'API de la bibliothèque standard de Java (dans ses différentes déclinaisons : *Standard Edition*, *Enterprise Edition* et *Micro Edition*) ;
- la spécification du *bytecode*, des fichiers `class` et du modèle d'exécution par machine virtuelle [15].

La compilation de code Java en *bytecode* ne correspond pas à une compilation « classique », car le compilateur produit un code intermédiaire (le langage de la machine virtuelle Java ou *bytecode*). Celui-ci n'est en général pas compréhensible directement par le microprocesseur. En fait, une partie des tâches incombant à un compilateur « classique » (non Java) est reportée sur un environnement d'exécution (JRE) implémentant la machine virtuelle Java (JVM). La tâche du compilateur Java est donc restreinte, puisque le fossé sémantique entre le format d'entrée du compilateur, les fichiers sources en Java, et le format de sortie, les fichiers `class` sous forme de *bytecode*, est faible. De plus, la spécification de Java impose de réaliser les vérifications d'intégrité avant exécution, via la verification de *bytecode*. La plupart des propriétés de sécurité du langage Java est donc assurée par l'implémentation de la JVM.

La spécification officielle de la JVM [15] ne spécifie pas pour autant un modèle d'exécution précis pour cette machine virtuelle :

> *[...] the Java virtual machine does not assume any particular implementation technology, host hardware, or host operating system. It is not inherently interpreted, but can just as well be implemented by compiling its instruction set to that of a silicon CPU. It may also be implemented in microcode or directly in silicon. [..]*
>
> *To implement the Java virtual machine correctly, you need only be able to read the class file format and correctly perform the operations specified therein. Implementation details that are not part of the Java virtual machine's specification would unnecessarily constrain the creativity of implementors. For example, the memory layout of run-time data areas, the garbage-collection algorithm used, and any internal optimization of the Java virtual machine instructions (for example, translating them into machine code) are left to the discretion of the implementor.*

La spécification laisse ainsi le choix du moyen d'implémentation mis en œuvre pour exécuter un programme au format *bytecode* Java, tant que ce modèle respecte la spécification officielle. Les seuls points imposés par la spécification sont :

- le format des fichiers `.class` et du *bytecode* ;
- le chargement, l'édition des liens et l'initialisation des classes ;
- les vérifications réalisées par le vérificateur de *bytecode* sur les fichiers `class` ;
- le comportement attendu pour l'exécution d'un programme *bytecode* Java (la sémantique du *bytecode*, via la spécification d'une machine abstraite à pile) ;
- la gestion des exceptions ;

– la gestion des accès concurrents et des *threads* (la spécification n'imposant aucune manière spécifique d'implémenter les *threads*, la gestion des *threads* ne recouvre l'implémentation de la classe des *threads* dans la bibliothèque standard).

La spécification mentionne également d'autres services que la JVM doit implémenter, mais pour lesquels elle fournit peu de détails :

– la gestion automatique des objets créés sur le tas par un mécanisme de glaneur de cellules (*garbage collector*). Un objet ne doit jamais être désalloué explicitement ;
– l'interface avec des classes de la bibliothèque standard pour implémenter certains services (par exemple, le chargement de classes, l'introspection ou la gestion des *threads*) ;
– les étapes de finalisation, de déchargement de classes et d'arrêt de la JVM.

Certains services, considérés comme standard, ne sont pas décrits dans la spécification de la JVM [15], mais dans d'autres documents :

– l'interface JNI permettant au code Java d'appeler des fonctions implémentées en C/C++ (et inversement d'initialiser une instance de la JVM depuis une application écrite en C/C++) ;
– l'interface JVMTI qui permet de contrôler la JVM et le flux d'exécution d'une application Java à des fins de mise au point.

Etant données les libertés offertes par la spécification de la JVM, notamment en ce qui concerne l'exécution du *bytecode*, les implémentations de la plate-forme d'exécution Java (JRE) diffèrent sensiblement. Ces implémentations s'appuient sur un format d'entrée (le *bytecode*, les fichiers `class`) et une sémantique définie précisément. En revanche, différentes stratégies ou modèles d'exécution peuvent être adoptés privilégiant la vitesse d'exécution, la portabilité, l'empreinte mémoire, etc. Ce document différencie trois grandes stratégies :

1. l'exécution à l'aide d'un émulateur ;
2. la compilation native statique ;
3. l'utilisation d'un processeur (ou d'un co-processeur Java) capable d'exécuter directement les instructions du *bytecode*.

Le terme JVM est souvent utilisé dans la littérature pour désigner des éléments différents. Dans la spécification de Sun [15], le terme désigne la spécification d'une machine à pile abstraite chargée d'exécuter le *bytecode*. Le terme est aussi généralement utilisé pour désigner une implémentation à l'aide d'un émulateur de cette spécification. Il est également parfois utilisé à tort pour désigner l'environnement d'exécution (JRE) qui comprend, outre l'implémentation de la JVM, une implémentation de la bibliothèque standard Java. En toute rigueur, les différents modèles d'exécution doivent, pour assurer la compatibilité avec le modèle d'exécution de Java, implémenter la machine virtuelle Java.

Le premier modèle correspond à une implémentation littérale de la JVM sous forme logicielle. Il s'agit d'un module exécutable (généralement une bibliothèque partagée) implémentant différents services dont l'exécution du *bytecode*, souvent à l'aide d'un interpréteur. Un compilateur « à la volée » (*JIT*) peut également être utilisé conjointement à l'interpréteur ou pour le remplacer. L'implémentation de la JVM joue alors pleinement son rôle d'interface entre l'application distribuée sous forme de *bytecode* et la plate-forme d'exécution native (nous supposons

pour ce type de modèle d'exécution que le processeur utilisé est un processeur générique qui ne peut exécuter le *bytecode* nativement). Il s'agit du modèle d'exécution le plus couramment utilisé, notamment par l'implémentation de référence de Sun du JRE. Il est donc considéré comme le modèle d'exécution standard.

Le second modèle repose sur une étape de compilation réalisée avant l'exécution (AOT, *Ahead Of Time*) de l'application et qui fournit un code natif pour l'ensemble des méthodes exécutées par l'application. Le code des bibliothèques Java utilisées par l'application est intégré à l'exécutable ou fourni sous la forme de bibliothèques natives partagées. La compilation peut être exécutée en deux étapes (Java vers *bytecode*, puis *bytecode* vers code natif) ou directement depuis le code source Java vers le code natif. Ce mode d'exécution permet ainsi de distribuer les applications sous la forme d'un exécutable natif autonome. Il n'est alors plus nécessaire d'installer un environnement d'exécution Java, mais ce modèle d'exécution s'écarte alors de la spécification de la JVM (qui impose notamment de distribuer les applications Java sous forme de *bytecode*).

Le dernier modèle s'appuie sur une implémentation (souvent partielle) de la JVM sous forme matérielle. Il s'agit généralement de processeurs ou de co-processeurs dont le jeu d'instructions implémente (généralement en partie) le jeu d'instructions de la JVM (les instructions du *bytecode*). Le *bytecode* ne comportant pas d'instructions permettant d'accéder aux ressources natives (par exemple, pour la gestion des entrées/sorties), ce type de processeur comprend également d'autres instructions permettant d'effectuer ces accès ou utilise conjointement un processeur « générique ». De plus, certains services de la JVM (par exemple, la gestion de la mémoire ou le chargement de classes) étant difficilement implémentables sous forme matérielle, ce modèle d'exécution s'appuie également sur une implémentation logicielle (généralement désignée sous le terme de JVM) qui fournit ces services et utilise le processeur dédié pour l'exécution du *bytecode*. Ce mode d'exécution reste aujourd'hui peu utilisé en raison du coût d'implémentation d'une plate-forme matérielle spécifique. Principalement employé pour des raisons d'optimisation, il est surtout mis en œuvre pour des applications embarquées et supporte parfois un sous-ensemble du langage Java et de la bibliothèque standard.

Cette étude identifie donc les différentes stratégies du modèle d'exécution de la JVM. Ces stratégies ont été regroupées selon trois grandes catégories, mais l'étude a fait apparaître que la frontière entre ces différentes catégories était parfois mince : un émulateur peut utiliser des techniques de compilation native pour optimiser l'exécution du *bytecode*, un compilateur statique peut embarquer un interpréteur dans l'exécutable, une architecture à base de processeurs Java peut nécessiter une implémentation logicielle de la JVM pour assurer certains services, etc.

4 MODÈLE D'EXÉCUTION PAR ÉMULATEUR

Nous abordons dans cette section le modèle d'exécution standard de Java. Celui-ci repose sur l'utilisation d'un émulateur qui implémente la JVM. Il s'agit de la stratégie d'implémentation la plus répandue (les principaux environnements d'exécution Java, dont l'implémentation de Sun, reposent sur cette stratégie) et la plus mature (les premiers environnements d'exécution Java, développés dans les années 1990, implémentaient cette stratégie). La spécification de la JVM n'impose aucune architecture précise en ce qui concerne son implémentation. Toutefois, il est possible de regrouper les services implémentés par un émulateur Java suivant différents blocs fonctionnels :

- le cœur de l'émulateur : il regroupe notamment les fonctions de gestion du démarrage et de l'arrêt de l'émulateur, le mécanisme de chargement et d'initialisation des classes, le mécanisme d'interface avec le code natif (JNI) ainsi que différents mécanismes utilitaires (JVMTI, audit, etc.) ;
- le vérificateur de *bytecode* (*Bytecode Verifier*) : élément clé de la sécurité des applications Java, il vérifie l'intégrité des fichiers `class` (et notamment leur typage) avant leur exécution ;
- le gestionnaire des fils d'exécutions : il assure la gestion (et notamment l'ordonnancement) des différents *threads* de la JVM (dont les *threads* Java imposés par la spécification du langage et de la JVM). Il assure également la synchronisation entre ces *threads* ;
- le système d'exécution : il est chargé d'exécuter le *bytecode* des classes chargées. Il implémente généralement un interpréteur, mais peut également avoir recours à un compilateur natif pour effectuer des optimisations ;
- le système de gestion de la mémoire : il est chargé de l'allocation et de la désallocation dynamique de la mémoire de l'émulateur. Il comprend notamment un système de représentation des classes en mémoire ainsi qu'un système de gestion des objets références (instances de classes et tableaux Java). La spécification de la JVM impose qu'une application Java ne puisse désallouer explicitement les objets. Le système de gestion de la mémoire comprend donc obligatoirement un mécanisme de glaneur de cellules (*garbage collector*) permettant de désallouer automatiquement les objets qui ne sont plus référencés ;
- l'interface avec la plate-forme native : ce bloc fonctionnel comprend notamment l'interface avec les mécanismes de l'OS natif (gestion des *threads*, des signaux, des entrées/sorties, etc.). Il peut également comporter (si l'émulateur est destiné à fonctionner sur différentes architectures matérielles) une couche d'abstraction de l'architecture matérielle.

Cette étude porte sur la préservation des propriétés de sécurité identifiées dans l'étude du langage Java [6] en fonction des différentes stratégies d'implémentation de la JVM. Ce document se focalise donc sur les principaux mécanismes de sécurité implémentés au sein d'un émulateur Java, ainsi que sur les mécanismes ayant un impact fort sur la sécurité des applications Java exécutées tout en s'appuyant sur l'architecture décrite précédemment. Les principes

de la compilation Java vers *bytecode* sont rappelés en section 4.1, puis les différents blocs fonctionnels d'un émulateur implémentant la JVM sont analysés :

1. La section 4.2 présente les mécanismes de démarrage des applications Java et de chargement de classes. Ces mécanismes ont principalement un impact sur la disponibilité et l'intégrité du code de l'application sous forme de fichiers `class`.
2. La section 4.3 aborde le mécanisme de vérification de *bytecode*. Il s'agit du principal mécanisme de sécurité de la JVM qui assure la correction des applications distribuées sous forme de *bytecode*.
3. La section 4.4 porte sur l'exécution du *bytecode*. Les mécanismes du système d'exécution doivent en effet implémenter certaines vérifications dynamiques qui permettent d'assurer la confidentialité et l'intégrité des données et du code de la plate-forme Java (en s'assurant notamment de l'absence de débordement de tampon). De plus, les mécanismes d'exécutions reposant sur une étape de compilation doivent assurer l'intégrité du code natif produit.
4. La section 4.5 expose les différentes stratégies d'implémentation des *threads* Java ainsi que la problématique de synchronisation. Le mécanisme de gestion des *threads* assure en effet la disponibilité des différents *threads*.
5. La section 4.6 présente le mécanisme de gestion de la mémoire et notamment le glaneur de cellules. Ce mécanisme possède un impact sur l'intégrité des données (et sur la confidentialité des données au travers de la problématique de rémanence des données confidentielles).
6. La section 4.7 évoque la problématique générale de l'intégration de l'émulateur avec les mécanismes de l'OS. Celui-ci joue notamment un rôle important dans le confinement entre les différentes applications Java.

Pour chacun des blocs fonctionnels étudiés, le fonctionnement des principaux mécanismes est rappelé brièvement, ainsi que les propriétés de sécurité assurées. Lorsque différentes implémentations partagent certaines stratégies d'implémentation d'un même mécanisme, celles-ci sont également rappelées et comparées en termes d'impact sur la sécurité, de maturité et de complexité [1]. Cette étude précise les « points sensibles » qui, s'ils font l'objet d'un défaut de conception ou d'implémentation, remettent en cause la sécurité de l'application. Ces points doivent donc faire l'objet d'une attention particulière, par exemple lors de l'évaluation d'un émulateur Java. Cette étude précise également les mécanismes ou stratégies d'implémentation qu'il est souhaitable d'intégrer à une implémentation d'une JVM de confiance et inversement, ceux qu'il convient d'éviter (ou de restreindre). Dans la plupart des cas, il s'agit d'établir un compromis entre le niveau de sécurité souhaité et l'optimisation du temps d'exécution.

1. En revanche, les détails propres à chaque implémentation ne sont pas détaillés. Ils pourront être abordés dans le rapport sur l'étude et la comparaison des JVM [5].

4.1 Principes de la compilation

La spécification de la JVM alloue un chapitre entier à la compilation de programme Java vers le format *bytecode*. Cette description est très informelle et principalement faite à partir d'exemples. Les transformations présentées sont directes, sans optimisation particulière. Les principes de la compilation Java vers *bytecode* ont été présentés dans le livrable [4]. La section 3.2 explique notamment le rôle limité du compilateur Java dans la préservation des propriétés de sécurité. Les vérifications effectuées par le compilateur ont essentiellement un rôle d'aide au développement en prévenant le développeur que son code n'est pas conforme et qu'il sera rejeté à l'exécution. Le risque majeur de la compilation et de changer le comportement attendu d'un programme, mais les optimisations limitées qui sont généralement employées limitent ce risque. Bien que rien ne semble officiellement dit à ce sujet, il semble que la pratique soit plutôt de réaliser une traduction directe du niveau source au niveau *bytecode* et de laisser à la JVM (à travers un *JIT* généralement) le soin d'optimiser le programme.

4.2 Démarrage de l'application Java, chargement de classes et initialisation

La spécification de la JVM définit précisément les différentes phases de démarrage d'une application Java (chargement de classes, éditions de liens incluant la vérification de *bytecode*, préparation et résolution, et enfin initialisation des classes). Ces différentes phases ont déjà été présentées dans la section 6.3 du « Rapport sur le langage Java » [6] en raison de leurs impacts sur le mécanisme de contrôle d'accès.

4.3 *Bytecode Verifier*

Les classes chargées par une machine virtuelle Java sont soumises à un processus de vérification. Cette vérification garantit que la représentation binaire d'une classe ou d'une interface est « structurellement correcte » (voir [5]). Si la vérification échoue, une exception `VerifyError` est levée. La réalisation de ce processus de vérification lors du chargement de classes permet d'améliorer les performances de l'exécution en libérant l'interpréteur des tests dynamiques correspondants (par exemple, l'absence de débordement des piles d'opérandes). Une machine virtuelle Java dispose de deux stratégies pour vérifier une classe. Les deux types de vérifications sont l'**inférence** et la **vérification** (au sens strict du terme) **par *stackmaps***. Dans le premier cas, la machine virtuelle calcule entièrement l'information nécessaire à la vérification de la conformité du *bytecode*. Dans le second cas, une information pré-calculée (typiquement par le compilateur) est incluse dans le fichier (approche *Proof Carrying Code*). La présence de cette information, appelée table de *stackmaps*, permet d'accélerer fortement le processus

de vérification. Selon la spécification de Sun, un fichier `class` dont le numéro de version est strictement inférieur à 50.0 doit être vérifié uniquement par inférence (un tel fichier ne contient normalement pas de *stackmaps*). Un fichier dont le numéro de version est strictement supérieur à 50.0 doit être vérifié uniquement à l’aide des *stackmaps*. Le cas des fichiers 50.0 est un peu particulier. La vérification à base de *stackmaps* doit être utilisée. Cependant, en cas d’échec de cette dernière, la vérification par inférence peut être utilisée. Ces deux mécanismes sont présentés dans [6, 5]. Les propriétés assurées par le vérificateur de *bytecode* sont déjà présentées dans [6].

4.4 Exécution de *bytecode*

Dans cette partie nous présentons les différentes techniques d’exécution de programme *bytecode*. Toutes s’appuient sur des optimisations plus ou moins complexes. Nous présentons les optimisations principales et leurs risques en termes de sécurité.

4.4.1 Interprétation

Dans le modèle d’émulateur standard, la plate-forme d’exécution charge un fichier `.class` contenant la classe principale du programme [2] (*c.f.* la section 4.2 sur le chargement de classe), puis commence l’exécution de la méthode principale (la méthode `public static void main (String[])` de cette classe).

Chaque instruction *bytecode* est décodée puis exécutée. L’allocation dynamique de mémoire, la libération automatique et la gestion des *threads* sont gérées par des appels à des services de la machine virtuelle. Cette approche a l’avantage de suivre directement la spécification officielle. Néanmoins, elle souffre d’un défaut de taille : la lenteur de l’exécution des programmes.

Plusieurs modes d’interprétation existent et se différencient de par leur complexité intrinsèque. Cette complexification est due à la volonté d’améliorer les performances et s’accompagne d’une distanciation vis-à-vis de la spécification. Les principaux modes d’interprétation sont les suivants :

– mode aiguillage (ou *switch*) : c’est le mode d’interprétation le plus proche de la spécification. Dans ce mode, les instructions *bytecode* d’une méthode sont analysées une à une et sont décodées à l’aide d’une opération conditionnelle de type *switch/case*, qui redirige vers un traitant spécifique. Ce traitant est ensuite exécuté. Ce mode d’interprétation, notamment utilisé dans JamVM et HotSpot (le *C++ interpreter*), a pour

2. L’identification de la classe principale n’est pas précisée dans la spécification de la JVM.

défaut d'être lent, du fait de la lourdeur de l'opération d'aiguillage ;

– mode *threaded* : dans ce mode, le test conditionnel *switch/case* est supprimé. A chaque instruction correspond un index dans un tableau contenant l'adresse de l'ensemble des traitants. Deux cas de figure sont alors possibles :
  – soit une indirection subsiste (mode *indirect threaded*) : une routine se charge d'identifier dans le tableau l'adresse du traitant associé à chaque instruction *bytecode* ;
  – soit l'instruction *bytecode* est directement remplacée par l'adresse du traitant (mode *direct threaded*). L'exécution consiste donc à enchaîner les traitants pointés par ces adresses. Ce mode d'interprétation ne pose pas de problèmes de compatibilité avec PaX car les traitants sont générés à la compilation de la JVM et pas dynamiquement lors de son exécution. En effet, seuls des tableaux d'adresses (un par méthode interprétée) de traitant dont les accès sont des lectures sont générés dynamiquement, ce qui n'alertent pas PaX.

  Ce mode est par exemple utilisé dans JamVM et HotSpot ;
– mode dépilé (ou *inline*) : dans ce mode, notamment utilisé dans JamVM, l'ensemble des instructions *bytecode* d'une méthode est remplacé par une liste chainée contenant le corps des traitants correspondants. Il s'agit donc d'une forme de compilation simplifiée, réalisée au moment du décodage des instructions. Cette opération n'est réalisée qu'une seule fois, lors du premier appel de la méthode. Ensuite, lors de l'exécution, il suffit uniquement d'enchaîner ces portions de code machine. En outre, ce mode nécessite que l'espace mémoire alloué pour stocker les instructions *bytecode* possède à la fois des droits en écriture (pour le remplacement des instructions par le corps des traitants) et en exécution (pour l'exécution des traitants). De fait, ce mode d'interprétation n'est pas compatible avec des protections mémoire telles que PaX ;
– mode *template* : ce mode, mis en œuvre dans HotSpot, est similaire au mode *inline*, à ceci près que le corps des traitants ne contient pas du code machine directement exécutable, mais une forme de méta-assembleur générique (c'est-à-dire, agnostique de la plate-forme sous-jacente). À partir de la liste contenant l'enchainement des traitants, ceux-ci sont compilés à la volée et exécutés un par un. Ce mode d'interprétation se rapproche grandement d'une compilation de type *JIT*, à cette différence près que l'opération est réalisée pour chaque instruction *bytecode*, contrairement au mode *JIT* où des optimisations sont réalisées sur l'enchaînement des instructions *bytecode* (réordonnancement, fusion ou encore suppression d'instructions). Ce mode d'interprétation n'est également pas compatible avec des protections mémoire telles que PaX.

Au final, plus le mode d'interprétation est optimisé, plus il s'écarte de la spécification de base et se rapproche d'une compilation *JIT*. La frontière entre le mode *template* et la compilation *JIT* est d'ailleurs très mince. En outre, les modes optimisés d'interprétation sont peu ou pas compatibles avec des protections mémoires avancées.

En ce qui concerne l'implémentation de la représentation de la pile d'opérandes [5], deux approches sont possibles : soit utiliser une pile native du système d'exploitation, soit utiliser

une pile gérée de manière *ad hoc* au-dessus du tas natif. La spécification laisse le libre choix quant au mode d’implémentation de la représentation de la pile. Une optimisation possible pour la pile d’opérandes consiste à mettre en œuvre un cache pour le haut de la pile. Il s’agit d’une structure très facilement accessible dans laquelle sont placées les opérandes manipulées par des traitants réalisant des opérations internes ou combinées. Un point important concerne la bonne synchronisation du cache de pile et de la pile elle-même.

4.4.2 Compilation « à la volée » (*JIT*)

Afin d’accélérer l’exécution du *bytecode*, il est tentant de compiler les instructions *bytecode* en code machine pour la plate-forme sous-jacente, puis d’exécuter directement ce code compilé. Réaliser cette compilation avant l’exécution (compilation « statique ») ou *Ahead-of-Time*, *c.f.* section 5) pose un problème de portabilité dans le modèle Java où les programmes ont vocation à être distribués : un code natif dépend de la plate-forme d’exécution et les vérifications statiques assurées par le vérificateur de *bytecode* sont très difficiles à réaliser sur ce format de programme. De plus, le langage Java est un langage « dynamique » : certaines classes à exécuter ne sont réellement connues qu’à l’exécution du programme, se trouvent parfois sur un réseau distant et ne peuvent donc pas être compilées avant l’exécution. Ce problème a motivé l’utilisation de la compilation « dynamique » de type *Just In Time Compiler* : la compilation a lieu au fur et à mesure de l’exécution. La première méthode à exécuter est compilée en code natif puis exécutée. Lors d’un appel de méthode, la classe correspondante est chargée (si ce n’est pas encore le cas), puis la méthode appelée est compilée et l’exécution continue ainsi. Là encore, l’allocation dynamique de mémoire, la libération automatique et la gestion des *threads* sont gérées par des appels à des services de la JVM.

Pour l’utilisateur, tout se passe comme si le programme Java, distribué sous forme de fichiers `class`, était interprété. En particulier, un programme est d’abord vérifié par le vérificateur de *bytecode* avant d’être compilé dynamiquement. Il reste à la charge du compilateur dynamique d’insérer les vérifications dynamiques nécessaires pour toutes les vérifications de typage qui ne sont pas du ressort du vérificateur de *bytecode* telles que la vérification des accès dans les bornes de tableaux, le déréférencement de pointeurs nuls, etc. (voir la section 5.4 de [6]).

Contrairement à la compilation statique, le temps de compilation d’un compilateur dynamique rentre en compte dans le temps d’exécution du programme. Cette contrainte particulière impose aux compilateurs dynamiques de trouver un juste compromis entre la compilation rapide mais peu optimisée et la compilation coûteuse mais accélérant significativement l’efficacité d’un programme. Pour être plus précis, la compilation dynamique de type *AOT* a uniquement un impact sur le temps de démarrage de l’application (cela peut être ou non une bonne stratégie, cela dépend des besoins). Si l’exécutable est persistent, l’impact n’est présent que lors de la première exécution. On est alors très proche d’une solution de type compilation statique ou la compilation est réalisée lors du développement, l’application étant par la suite déployée sous forme binaire sur le système.

Lorsque le compilateur dynamique cherche à effectuer des optimisations complexes, cela peut provoquer un délai important au démarrage de l'application. La JVM Hotspot propose en conséquence deux compilateurs distincts à choisir en fonction du type d'application exécutée : le compilateur `client` propose un démarrage rapide des applications grâce à des optimisations peu agressives, tandis que le compilateur `server` met en œuvre des techniques d'optimisations avancées (similaires aux optimisations C++ standards), au prix d'un démarrage plus lent.

4.4.3 Interprétation et optimisation dynamique du *bytecode*

Le surcoût de temps d'exécution lié au temps d'optimisation est une limitation intrinsèque de l'approche *JIT* qui compile dynamiquement (et optimise légèrement) **toutes** les méthodes appelées durant l'exécution. Les JVM actuels (comme Hotspot de Sun) suivent de ce fait une approche plus hybride. Durant l'exécution, les premières méthodes ne sont pas optimisées, voir pas compilées du tout. Cela assure un démarrage relativement rapide. Au cours de l'exécution, la plate-forme pourra décider qu'une méthode ou une portion de méthode nécessite une compilation native et une optimisation avancée, car cette portion semble être fréquemment utilisée. En fonction du degré d'importance estimée pour cette portion, le compilateur lance une optimisation plus ou moins poussée en faisant le pari que le temps « perdu » à optimiser sera ensuite gagné par l'accélération de cette portion critique. La détection des « points chauds » nécessite généralement de compter le nombre de passage dans les méthodes, voir dans les têtes de boucle. Le passage incessant entre mode interprété et mode compilé est *a priori* complexe, mais il est de toute façon nécessaire à tout interpréteur pour appeler du code natif.

4.4.4 Optimisations réalisées et contraintes sémantiques

La technique de détection des points chauds a un impact faible sur la sécurité. Une mauvaise détection ne fait en général que ralentir l'exécution. Elle n'a d'impact réel que si le *JIT* possède des vulnérabilités. En revanche, les optimisations réalisées par le *JIT* peuvent quant à elles modifier significativement le comportement attendu en cas d'erreurs de conception de ces optimisations. Si le *JIT* présente une telle erreur, il est *a priori* possible pour un attaquant de « profiler » son code malveillant de manière à ce qu'il soit compilé et optimisé par le *JIT*, ce qui lui permet d'exploiter la vulnérabilité en question.

Nous listons maintenant les principaux aspects du langage Java qui font généralement l'objet d'optimisations :

- le dépliage des méthodes qui consiste à remplacer un appel de méthode par le code correspondant ;
- l'optimisation des vérifications dynamiques intrinsèques au langage ;
- la suppression des verrous inutiles pour la synchronisation.

Ensuite, nous discuterons des annotations utilisées par certaines techniques de *JIT* pour accélérer leurs recherches d'optimisations. Enfin, nous terminerons cette partie en présentant une liste non-exhaustive d'optimisations classiques (non spécifiques aux langages orientés objet).

*4.4.4.1 Dépliage de méthode (*inlining*)*

L'optimisation la plus importante est le dépliage des appels de méthodes. La programmation orientée objet incite à écrire de nombreuses méthodes relativement petites en taille (les méthodes classiques de lecture d'un champ privé par exemple). Cela donne lieu à de nombreux appels de méthodes, ce qui se traduit par un surcoût à l'exécution lié au changement de contexte. Une technique d'optimisation standard consiste à déplier l'appel. C'est une transformation importante pour les autres optimisations tentées par le compilateur, car la plupart d'entre elles sont « intra-procédurales », c'est-à-dire qu'elles travaillent méthode par méthode. Cependant, le dépliage n'est pas toujours applicable avec les appels virtuels où la méthode réellement appelée n'est pas facilement identifiable. Par exemple, un appel virtuel à la méthode `A.m()` au niveau *bytecode* peut effectivement mener à l'appel d'une méthode `m` déclarée dans la classe `A`, mais aussi à une des surcharges de cette méthode déclarée dans une classe qui hérite de `A`.

L'exemple suivant montre un cas où il est facile de prévoir que l'appel virtuel `m()` (noté `A.m()` au niveau *bytecode*) se fera sur un objet de classe `A`.

```
A a = new A();
a.m();
```

Dans ce deuxième exemple, il est beaucoup plus difficile de prévoir quelle méthode sera effectivement appelée à cause de l'utilisation du polymorphisme objet.

```
void m(Object x) {
   ...
   x.toString();
   ...
}
```

La méthode réellement appelée n'est généralement connue qu'à l'exécution. Néanmoins, les expériences montrent que la plupart des appels virtuels d'un programme n'ont pour cible qu'une seule méthode. Les compilateurs mettent donc en œuvre de nombreux efforts pour réussir à « dévirtualiser » ces appels, c'est-à-dire prédire quelle méthode sera effectivement appelée. Le compilateur Hotspot de Sun utilise même parfois une prédiction optimiste : lorsqu'un appel virtuel est rencontré à l'exécution, la méthode effectivement appelée (connue à l'exécution) est dépliée en espérant qu'au prochain passage sur cet appel, la même méthode sera appelée. Si le prochain appel virtuel au même point de programme appelle effectivement une autre méthode, le dépliage précédent est annulé. Un tel compilateur doit donc être capable de défaire certaines de ses optimisations.

Un dépliage incorrect de méthode peut avoir des conséquences indirectes sur la sécurité d'un programme, puisqu'il peut amener à l'appel d'une mauvaise méthode. Par exemple, une classe `A` peut avoir redéfini une méthode `clone()` qui assure une copie sécurisée des instances. Si le compilateur utilise une optimisation de dépliage incorrecte, il se peut qu'il copie un objet de classe `A` avec la méthode définie par défaut dans la classe `java.lang.Object` au lieu d'utiliser la méthode `clone()` définie dans `A`.

4.4.4.2 Vérifications dynamiques et gestion des exceptions

Les propriétés intrinsèques du langage Java reposent en partie sur des vérifications dynamiques (pas d'accès en dehors des bornes des tableaux, par exemple). Lorsqu'une de ces vérifications échoue, la plate-forme d'exécution doit lancer une exception adéquate qui peut éventuellement être rattrapée. Le compilateur doit assurer que les variables locales sont disponibles au point de rattrapage. Il doit aussi assurer que les exceptions soient lancées dans le même ordre que celui imposé par une exécution standard. Cela gêne particulièrement les optimisations qui cherchent à réordonner le code.

Par exemple pour optimiser une portion de code telle que :

```
while (k<n) {
  ...
  x[k] = t[i] + t[i+1];
}
```

le compilateur va chercher à insérer un test `0<=i && i+1 < t.length` avant l'entrée de la boucle. Il lui faut pour cela détecter que `i` ne varie pas dans la boucle et que `t` désigne toujours un tableau de même taille. Il lui faut aussi s'assurer que la portion `...` ne lance pas d'exception, sans quoi une inversion dans l'ordre des exceptions lancées pourrait se produire. Cette inversion peut mener au masquage d'une exception de sécurité.

Une dernière vérification dynamique qui fait généralement l'objet de nombreuses optimisations est le test de sous-typage. Cette opération clé peut être intensivement utilisée durant l'exécution d'un programme (pour les opérations de *cast*, les affectations d'éléments de tableaux ou bien les manipulations d'interfaces). Les techniques d'optimisations généralement utilisées essayent de tabuler l'opération (mais la mise à jour de la tabulation [3] est nécessaire lors d'un chargement dynamique de classes), ou bien de mémoriser les résultats des tests pour épargner les tests ultérieurs. Il s'agit d'une opération critique, car la relation de sous-typage est un socle de base pour la sécurité d'un programme Java.

3. Dans le cadre de la programmation dynamique, la tabulation consiste à mémoriser toutes les entrées/sorties possibles pour une opération dans une table.

4.4.4.3 Synchronisation

Les méthodes et les blocs synchronisés permettent d'assurer l'atomicité de l'ensemble des opérations d'une région. Ce mécanisme occasionne néanmoins une prise de verrous coûteuse à l'exécution. La réutilisabilité du code Java multiplie cependant les cas où la prise de verrou est inutile. Pour assurer une bonne synchronisation dans tous les cas, de nombreuses classes de la bibliothèque standard multiplient les prises de verrous. Lorsque les objets verrouillés ne sont pas partagés entre *threads*, le mécanisme est cependant inutile. Ce cas se produit souvent en pratique.

Des analyses « d'échappement » sont généralement utilisées (par exemple dans la JVM Hotspot de Sun) pour essayer de détecter les verrous inutiles. Le but est alors d'assurer qu'un objet reste local au *thread* courant. Dans l'exemple suivant le vecteur `v` n'échappe pas à la méthode `m` et les verrous des méthodes `add` de la classe `Vector` sont inutiles.

```
public String m() {
  Vector v = new Vector();
  v.add("1");
  v.add("2");
  v.add("3");
  v.add("4");
  v.add("5");
  return v.toString();
}
```

Une erreur de conception dans une optimisation de cette nature pourrait provoquer une mauvaise synchronisation d'un programme avec deux conséquences majeures :

- tout d'abord la perte des invariants internes d'une classe (comme `Vector`) dont les invariants internes sont normalement assurés par des mises à jour en région atomique ;
- mais aussi l'apparition de *data races* (voir [6]) qui peuvent faire perdre la cohérence séquentielle d'un programme.

Dans le premier cas, on considère le problème de l'atomicité des modifications dans une classe. Par exemple, si une classe contient deux champs dont la valeur du second dépend toujours de la valeur du premier champ et si on ne synchronise pas la modification de ces deux champs au sein de la classe, il serait alors possible à partir d'un autre *thread* d'accéder à ces valeurs et de n'y voir aucune dépendance.

Le second cas porte sur l'apparition de *data races*. Dans ce cas, il ne s'agit plus de « casser » les invariants d'une classe, mais l'absence de synchronisation sur certaines opérations peut compromettre l'ordre séquentiel du programme (voir la section 6.1.1.4 de [6] sur le réordonnancement des instructions par le compilateur).

4.4.4.4 Annotations pour l'optimisation

Pour réduire le temps passé à chercher des optimisations sans réduire pour autant la qualité de ces optimisations, certaines approches [9] proposent de réaliser des optimisations avant l'exécution et de transmettre des annotations pour guider le compilateur dynamique. Une annotation dans un fichier `.class` peut ainsi indiquer qu'un certain accès au tableau ne nécessite pas de vérification dynamique. Si certaines informations assurées par annotation ont un impact faible sur la sécurité, d'autres peuvent être extrêmement dangereuses. On trouve dans la première catégorie, les annotations chargées d'aider la détection des points chauds. Une mauvaise annotation insérée par un attaquant ne peut ici mener qu'à une perte de performance [4]. L'exemple de l'accès au tableau rentre lui dans la deuxième catégorie, celle qui affecte la sémantique d'un programme. Des annotations malicieuses peuvent ici compromettre les nombreuses propriétés du langage Java qui doivent être assurées dynamiquement. La seule alternative sûre est de ne pas croire « sur parole » de telles annotations, mais de les faire vérifier par le *JIT*. La technique se rapproche des techniques de code porteur de preuve (elle est présentée dans [6]) et est déjà utilisée par le vérificateur de *bytecode* pour le typage standard. À noter que la généralisation de l'approche code porteur de preuve a aussi l'avantage de réduire la complexité des mécanismes d'optimisation sur lesquels peut reposer la correction d'une machine virtuelle optimisée, car les vérificateurs d'annotations sont généralement de nature plus simple que les outils qui génèrent ces annotations.

4.4.4.5 Optimisations classiques non-spécifiques à Java

Dans les paragraphes précédents, nous n'avons mentionné que les optimisations spécifiques aux langages orientés objet. Voici une liste non-exhaustive d'optimisations qui sont appliquées non seulement à Java, mais également à d'autres langages non orientés objet :

- l'élimination de code mort ;
- le réordonnancement des instructions. Des instructions peuvent être réordonnancées pour pouvoir utiliser au mieux le *pipeline* du processeur, s'il n'y a pas de dépendance entre elles. Ce réordonnancement peut occasionner des « surprises », si les instructions peuvent lever des exceptions ;
- la propagation de constantes. Cela consiste à propager la valeur d'une variable lorsque cette valeur est une constante. Ceci permet de simplifier les calculs et/ou d'utiliser moins de registres ;
- l'extraction des boucles de calculs invariants. Si, dans une boucle, la valeur d'un calcul ne change pas d'une itération à l'autre, alors il peut être extrait de la boucle ;
- la propagation de copies. Cela consiste à supprimer les instructions qui copient la valeur d'une variable/registre dans une/un autre. Cette optimisation peut être effectuée lors de l'allocation de registres.

4. Si la perte de performance est trop importante, cela peut éventuellement mener à une forme de déni de service.

- le déroulage de boucle. Cela consiste à dupliquer le corps de la boucle au sein de celle-ci et de faire moins d’itérations, afin d’augmenter le taux de *cache hit* ;
- la distribution de boucle. Les instructions d’une boucle sont réparties dans plusieurs boucles ayant le même espace d’itération que l’original. Cette transformation est utilisée pour extraire des calculs parallèles d’une boucle séquentielle ;
- la fusion de boucles. Les instructions de deux boucles sont fusionnées dans une seule boucle. Cela permet par exemple d’améliorer les réutilisations de données ;
- les optimisations *peephole* (à lucarne). Ces techniques consistent à examiner des petites séquences d’instructions (la lucarne) et à essayer de les remplacer par une séquence plus efficace, comme par exemple en éliminant des instructions redondantes, en optimisant le flot de contrôle, ou en simplifiant les expressions algébriques.

4.4.5 Frontière statique/dynamique - Problématique de stockage et d’intégrité du code natif

La frontière entre compilation statique et dynamique est souvent mince. La compilation *AOT* est parfois utilisée par un émulateur pour optimiser un programme juste avant l’exécution. Le compilateur effectue une résolution statique, compilant ainsi la quasi totalité de l’application. Concernant les problèmes de portabilité, la distribution est toujours effectuée sous forme de *bytecode* et le vérificateur de *bytecode* peut remplir son rôle standard.

L’utilisation d’un compilateur (*JIT* ou *AOT*) suppose la production d’un code natif exécutable qui doit être stocké (au moins temporairement) puis exécuté. La problématique majeure en termes de sécurité consiste à assurer l’intégrité de ce code exécutable entre l’étape de compilation et son exécution. Une modification malicieuse permettrait de contourner les vérifications effectuées par le vérificateur de *bytecode*. Assurer cette intégrité est plus ou moins complexe suivant la persistance de ce code. En effet, certains compilateurs peuvent mettre en cache le code compilé pour limiter le nombre de compilation. Ce cache peut même, dans certains cas, être partagé entre plusieurs instances de l’émulateur. Se pose alors le problème de la taille de la base de confiance et du cloisonnement entre différentes instances de l’émulateur.

Remarque 1

L’intégrité du bytecode *doit être garantie après le chargement et la vérification par le vérificateur de* bytecode*, car le vérificateur de* bytecode *ne sera pas rappelé si le* bytecode *est modifié après son passage.*

4.5 Gestion des *threads*

4.5.1 *Threads* OS

Les *threads* sont un concept intéressant qui existe aussi bien dans le monde Java (il s'agit d'une exigence de la spécification du langage et de la JVM) que dans la plupart des OS sous-jacents. On parle alors respectivement de *threads* Java et de *threads* OS.

Sous UNIX, les *threads* OS sont définis par la norme *POSIX.1c, Threads extensions*, qui est également un standard IEEE (IEEE Std 1003.1c-1995).

Cette norme définit les fonctionnalités suivantes :

- les mécanismes de création, d'arrêt et de contrôle des *threads* ;
- la synchronisation *inter-threads* : mutex et variables de conditions ;
- des règles d'intégration dans le système UNIX.

En première approche, il est clair que les primitives fournies par les *threads* OS peuvent être facilement exploitées pour implémenter les *threads* Java. En pratique, c'est bien souvent l'architecture retenue par les implémentations de la JVM, qui se basent exclusivement sur les *threads* OS pour implémenter les *threads* Java.

Sous Linux plus particulièrement, les *threads* OS sont implémentés par la bibliothèque NPTL (*Native Posix Thread Library*).

4.5.2 Implémentation des *threads* Java par la JVM sans utilisation des *threads* OS

Une approche utilisée par les anciennes implémentations de la JVM (y compris celles de Sun en version 1.0), était de ne pas utiliser les *threads* OS pour implémenter les *threads* JVM. Cette approche est appelée *Green thread*. L'avantage principal concerne la portabilité de l'émulateur qui peut notamment être exécuté sur des OS qui n'implémentent pas de mécanismes de *thread*. En ne dépendant pas des *threads* OS, on assure également que l'implémentation de la JVM aura rigoureusement le même comportement quel que soit l'OS sous-jacent utilisé. Cette approche va généralement de pair (pour des raisons de facilité d'implémentation) avec une gestion collaborative des *threads* Java. Cela signifie que chaque *thread* doit collaborer avec les autres *threads* afin de leur passer la main périodiquement. Si un *thread* ne le fait pas correctement, il peut monopoliser l'accès au *CPU* au détriment des autres *threads*.

Cette approche souffre des inconvénients suivants :

- gestion collaborative du temps *CPU* ;
- forte contention au niveau des entrées-sorties. En effet, si un *thread* est bloqué sur une opération d'écriture ou de lecture de fichier, l'OS peut suspendre la JVM jusqu'à

ce que la requête d'entrée-sortie soit satisfaite, même si d'autres *threads* aimeraient avoir du temps *CPU* ;
- peu performante, car du point de vue OS, un seul fil d'exécution est utilisé. Les architectures de machines à fils d'exécution multiples (multi-cœurs ou multi-processeurs) ne peuvent pas être exploitées à leur plein potentiel.

Vu le nombre de ses inconvénients, il est donc peu surprenant que cette approche ne soit plus utilisée dans les JVM industrielles d'aujourd'hui.

4.5.3 Implémentation des *threads* Java par la JVM avec utilisation des *threads* OS

L'approche utilisée par la plupart des implémentations des JVM aujourd'hui repose sur les *threads* OS pour implémenter les *threads* Java.

Cette approche peut être déclinée suivant différentes stratégies :
- L'implémentation peut utiliser un mapping $N : M$. Dans ce cas de figure l'émulateur exécute N *threads* Java à l'aide de M *threads* OS. En général, on aura $N \geq M$ à un instant donné [5]. Cette approche a principalement pour but d'améliorer les performances sur les OS dont la création de *threads* est une opération gourmande (tel que Linux avant la version 2.6).
- L'implémentation peut utiliser un mapping 1 : 1. Dans ce cas de figure, l'émulateur repose sur une correspondance 1 pour 1 entre les *threads* Java et les *threads* OS. Cette approche est celle qui est la plus couramment utilisée aujourd'hui du fait de sa grande simplicité. Elle est notamment utilisée par l'implémentation de Sun de la JVM.

4.5.4 *Threads* de la JVM

Il est intéressant de noter qu'en plus d'utiliser les *threads* OS pour l'implémentation des *threads* Java, une implémentation de la JVM peut être amenée à instancier d'autres *threads* OS pour ses besoins propres. En particulier :
- le glaneur de cellules peut être exécuté concurentiellement dans un *thread* à part ;
- pour les implémentations qui supportent le *JIT* (cf section 4.4.2), la JVM peut instancier un ou plusieurs *threads* de compilation.

Une part importante de la tâche d'ordonnancement de l'implémentation de la JVM consiste à synchroniser les *threads* applicatifs avec les *threads* utilisés pour le fonctionnement interne de la JVM. En effet, certains mécanismes (glaneur de cellules, compilateur, etc.) nécessitent parfois que l'ensemble des *threads* Java de l'application soient stoppés afin de s'assurer que l'état de l'application n'évolue pas.

5. Il est envisageable pour une JVM de faire du « recyclage » de *threads*, ou alors de préallouer des *threads* par avance, ce qui pourrait amener à une situation où il y a plus de *threads* OS que de *threads* Java

Dans ce cas, la JVM « reprend la main » sur les *threads* Java en leur envoyant un signal UNIX d'interruption, typiquement retourné par la fonction `pthread_kill()`.

4.5.5 Implémentation des mécanismes de synchronisation Java

Il existe deux familles de primitives de synchronisation en Java :

- les primitives de synchronisation historiques, basées sur l'utilisation du mot-clé `synchronized`. Ces primitives se traduisent par l'utilisation au niveau *bytecode* des directives `monitorenter` et `monitorexit` ;
- les primitives de synchronisation de haut niveau du *package* `java.util.concurrent`. Ces primitives ne sont pas basées sur des instructions *bytecode*, mais sur des appels bas niveau JNI, situés dans `sun.misc.Unsafe`.

4.5.5.1 Implémentation du mot-clé `synchronized`

Les instructions *bytecode* `monitorenter` et `monitorexit` sont implémentées en utilisant un double mécanisme de verrouillage :

- Le premier est appelé *thin-locking*. Celui-ci permet d'acquérir un verrou dans le cas courant où le verrou n'est pas encore acquis par un *thread* concurrent. Ce mécanisme est basé sur l'utilisation d'une zone mémoire de la structure de l'objet, qui contient un numéro identifiant le *thread* de manière unique si le verrou est pris et 0 si le verrou n'est pas pris. L'opération de prise de verrou, afin d'être sure, nécessite de réaliser une comparaison et la prise de verrou à proprement parler en une seule opération atomique. Sur l'architecture considérée, ceci est implémenté en utilisant l'instruction x86 `lock`, suivie de l'instruction `cmpxchg`. L'avantage de ce mécanisme de *thin-locking* est qu'il est beaucoup plus rapide que le deuxième. Son inconvénient est qu'il ne permet pas d'endormir le *thread* courant si le verrou demandé est déjà pris ;
- Le deuxième mécanisme, lorsque le *thin-locking* expliqué ci-dessus ne peut fonctionner (verrou déjà acquis par un autre *thread*), est basé sur l'utilisation de mutex POSIX (fonctions `pthread_mutex_lock()`, `pthread_mutex_unlock()`, `pthread_mutex_trylock()`).

4.5.5.2 Implémentation des primitives de synchronisation du package `java.util.concurrent`

Ces primitives ne sont pas basées sur des instructions *bytecode*, mais sont implémentées en Java pour la plus grosse part, avec des appels à des fonctions natives exportées dans la classe `sun.misc.Unsafe` pour ce qu'il est impossible de traiter en Java.

Ces fonctions natives couvrent les points suivants :

- les opérations atomiques de comparaison et d'affectation (méthodes `compareAndSwapInt`, `compareAndSwapLong`, `compareAndSwapObject`), qui, comme dans le cas de l'implémentation du mot-clé `synchronized` explicitée ci-dessus, sont basées sur le couple d'instructions x86 `lock` puis `cmpxchg` ;
- les opérations de mise en pause/relance de *threads* : ce sont les méthodes `park` et `unpark`. Ces méthodes sont basées sur la prise/relâchement d'un mutex POSIX.

4.5.6 Conclusion

La liberté laissée à une implémentation peut paraître grande en ce qui concerne les choix de conception pour l'implémentation de la programmation concurrente, cependant en pratique l'architecture présentée ici est celle utilisée par les différentes JVM.

4.6 Gestion de la mémoire

4.6.1 Problématique de l'allocation

Il existe des appels système de l'OS qui permettent de préciser que certaines pages mémoire ne doivent pas être utilisées par le mécanisme de pagination (ou *swap*), c'est-à-dire être écrites sur le disque dur lors de la saturation de la mémoire vive. Un tel mécanisme n'est clairement pas souhaitable si l'application manipule des données sensibles telles que du matériel cryptographique. Ces données ne doivent en effet pas se retrouver sur un support de stockage de masse, pour des raisons évidentes de confidentialité. Les appels système en question ne sont cependant pas accessibles à l'intérieur d'une application Java. Or, il serait pertinent de pouvoir les utiliser au sein du processus du *garbage collector* afin d'obtenir une gestion sécurisée du tas.

La spécification de la machine virtuelle définit plusieurs zones de données en mémoire. On peut en distinguer deux types selon qu'elles sont locales aux *threads* ou globales.

4.6.1.1 Zones mémoire locales

À chaque *thread* est associé un registre `pc` et une pile d'appels.

- Si la méthode courante n'est pas native, le registre `pc` contient l'adresse de l'instruction en cours d'exécution. Dans le cas contraire, la valeur de ce registre est non définie ;

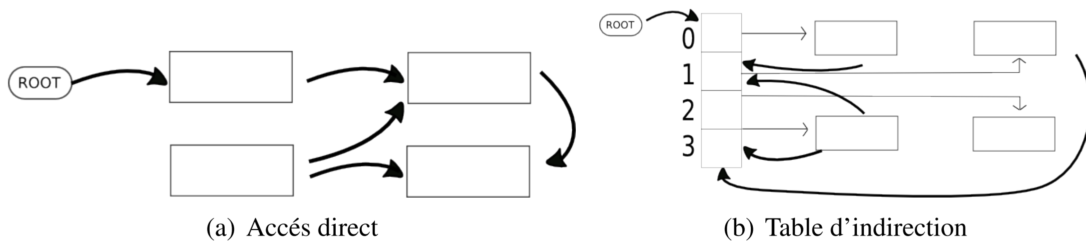
– La pile d'appels d'un *thread* contient des *frames* qui interviennent dans la gestion des variables locales, de la pile d'opérandes et des appels et retours de méthodes (voir [6], chapitre 5). La spécification autorise les piles d'appels à être allouées dans le tas (voir ci-dessous). Leurs tailles peuvent être fixes ou variables. Selon la spécification de la JVM, dans le cas de piles de taille fixe, les piles des différents *threads* n'ont pas nécessairement la même taille, mais rien n'indique qui décide de la taille. Dans le cas de piles de taille variable, l'utilisateur peut spécifier des tailles minimales et maximales. Le support des méthodes natives suppose également la présence d'une zone mémoire pour l'allocation des piles natives.

4.6.1.2 Zones mémoire globales

– Le tas est la zone mémoire, créée au démarrage de la machine virtuelle, où les instances de classe et les tableaux sont alloués. L'espace utilisé pour l'allocation d'objets est récupéré par un *garbage collector* dont le type n'est pas imposé par la spécification. La taille de cette zone mémoire peut être fixe ou variable et n'est pas nécessairement contiguë. L'utilisateur peut spécifier la taille de cette zone ou des valeurs minimales et maximales si elle est variable ;
– La zone de méthodes est la zone mémoire créée au démarrage de la machine virtuelle. Cette zone fait partie du tas, mais la gestion automatique de la mémoire sur cette zone est optionnelle. Comme le tas, sa taille peut être fixe ou variable et spécifiée par l'utilisateur. Cette zone mémoire contient les structures associées aux classes :
  – le *constant pool*, alloué dans la zone de méthodes, contient différentes constantes connues à la compilation (voir [6], chapitre 5),
  – la description (les signatures) des champs et méthodes,
  – le code des constructeurs et des méthodes.

4.6.1.3 Implémentation des références

La spécification de la machine virtuelle Java n'impose aucune contrainte concernant l'implémentation des références. On distingue généralement deux approches : l'accès direct et la table d'indirections (*handles*). Dans le second cas, le programme manipule les positions de la table plutôt que les adresses des objets (voir figure 1). L'intérêt d'une table d'indirections est que lorsque la position d'un objet en mémoire change (comme c'est le cas pour certains algorithmes de *garbage collection*, voir section 4.6.2), seule la table nécessite une modification. Les autres objets pointant vers cet objet ne sont pas affectés. L'inconvénient est que le déréférencement des objets devient plus coûteux.



(a) Accés direct (b) Table d'indirection

FIGURE 1 – Références

4.6.2 Problématique de la libération

Le rôle d'un *garbage collector* est de détecter parmi les blocs mémoire alloués par le programme ceux qui sont devenus inaccessibles. En Java, un bloc mémoire (ou objet en l'occurence) est accessible si :

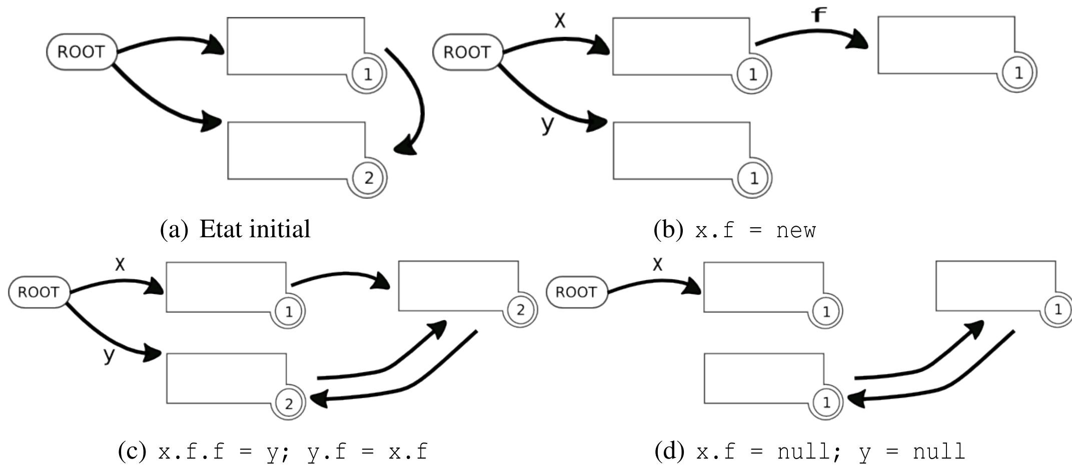
- la pile d'appels d'un *thread*, ou « racine » (*root*), contient une référence vers cet objet au travers d'une variable locale ou d'une pile d'opérandes ;
- un des champs d'un objet accessible pointe vers cet objet.

Plusieurs stratégies peuvent être utilisées pour l'implémentation d'un *garbage collector*. Les deux principales familles sont le comptage de références (*Reference Counting Collectors*, paragraphe 4.6.2.1) et l'exploration (*Tracing Collectors*, paragraphe 4.6.2.2). Le choix d'une stratégie dépend fortement du cahier des charges de l'application utilisant la gestion automatique de la mémoire. Un *garbage collector* peut être **exact** ou **conservatif**. Dans le premier cas, les références sont parfaitement identifiées et tous les blocs de mémoire non accessibles peuvent être libérés. Dans le second cas, il peut y avoir confusion entre les valeurs de type primitifs et les références. S'il n'est pas possible de distinguer ces deux types de valeurs, alors les premières seront interprétées par un *garbage collector* conservatif comme des références, ce qui peut conduire à considérer comme accessible un bloc mémoire qui ne l'est pas.

4.6.2.1 Comptage de références

La gestion automatique de la mémoire par comptage de références est le mécanisme le plus simple à mettre en œuvre. Chaque objet alloué en mémoire est associé à un compteur dénotant le nombre de références, vers cet objet, détenues par le programme. Si la valeur de ce compteur est 0, la mémoire occupée par l'objet peut être libérée. Le traitement des compteurs est fait par une instrumentation des instructions (figure 2). On suppose qu'il n'existe qu'une seule classe munie d'un unique champ `f`. La figure 2(a) représente l'état initial de la mémoire : deux objets sont accessibles depuis la racine et le premier pointe vers le deuxième. Les compteurs des deux objets ont donc respectivement pour valeur 1 et 2. Le champ du premier objet est mis à jour pour pointer vers un nouvel objet alloué par l'instruction `new` (figure 2(b), le nom de la classe est implicite). Le compteur du nouvel objet, pointé par le premier, est initialisé à la valeur 1. L'instrumentation de l'écriture décrémente le compteur du deuxième objet qui n'est plus pointé

par le premier. L'instrumentation de chaque instruction permet de libérer un objet dès que celui-ci devient inaccessible. Le coût de la gestion automatique de la mémoire est ainsi réparti dans le temps ce qui peut être crucial pour certaines applications (par exemple pour les applications temps réel). En revanche, la définition même du comptage de référence pose un problème pour la gestion des données cycliques. Ce problème est illustré par les figures 2(c) et 2(d) où un cycle est créé entre les objets `x.f` et `y`. Bien que ces objets deviennent inaccessibles (figure 2(d)) depuis la racine, leurs compteurs ont la valeur 1 ce qui empêche leur désallocation.



(a) Etat initial (b) `x.f = new`

(c) `x.f.f = y; y.f = x.f` (d) `x.f = null; y = null`

FIGURE 2 – Comptage de références

- \+ Technique très simple,
- \+ Libération de la mémoire immédiate,
- \+ Coût de l'exécution réparti dans le temps,
- \- Technique intrusive et coûteuse (instrumentation du code),
- \- Problème de gestion des données cycliques.

Remarque 2

Le problème des données circulaires peut être contourné en combinant le comptage de références avec un autre algorithme. Dans ce cas, ce dernier sera appelé occasionnellement pour pallier au problème. C'est par exemple le cas pour le langage Python.

4.6.2.2 Exploration

Les algorithmes à base d'exploration reposent sur un parcours de la mémoire permettant de découvrir, à partir des racines, les blocs mémoire accessibles par le programme. Tous ces algorithmes supposent la suspension du programme pendant leur exécution.

Mark and Sweep L'algorithme *Mark and Sweep*, constitué de deux phases, est la forme la plus simple de gestion automatique de la mémoire par exploration. La première phase (*Mark*) marque les références découvertes par l'exploration (marquage direct du bloc mémoire dans le tas). Ces éléments sont marqués au cours d'un parcours des éléments accessibles à partir de la racine (figure 3(b)). La seconde phase (*Sweep*) parcourt l'intégralité du tas et libère les blocs mémoire non-marqués (figure 3(c)). Le principal inconvénient de cet algorithme réside dans le parcours intégral du tas effectué lors la seconde étape. Même si peu de blocs mémoire sont effectivement utilisés par le programme, le temps d'exécution est proportionnel à l'espace réservé pour le tas.

ROOT

(a) Etat initial

ROOT

(b) Mark

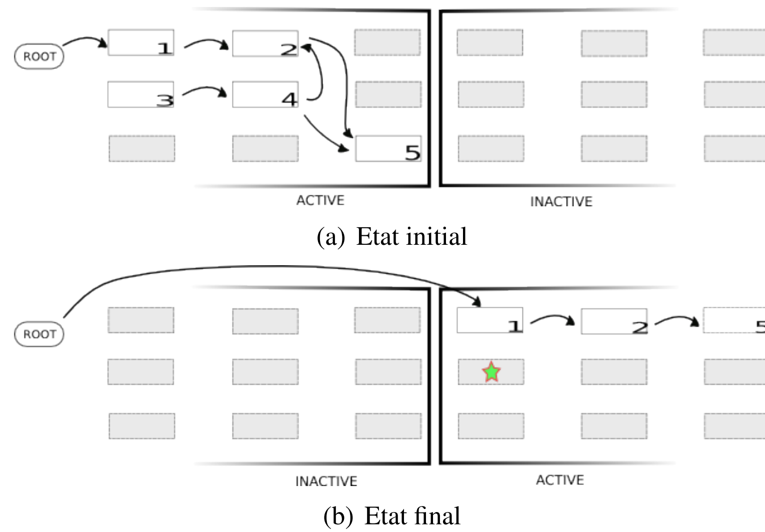
ROOT

(c) Sweep

FIGURE 3 – *Mark and Sweep*

+ Technique simple,
+ Technique peu intrusive,
+ Gère les structures de données cycliques,
- Parcours complet du tas (en particulier des objets inaccessibles).

Copying L'algorithme *Copying* permet d'éviter le parcours intégral du tas réalisé par le *Mark and Sweep*. Il ne comporte qu'une phase, mais nécessite en revanche l'utilisation de deux tas : un tas actif et un tas inactif. Comme précédemment, un parcours du tas (actif) à partir des racines est effectué pour découvrir les éléments accessibles (figure 4(a)). Au fur et à mesure que les éléments sont découverts, ils sont recopiés dans le tas inactif. Une fois le parcours achevé, les rôles des deux tas sont inversés (figure 4(b)). Le déplacement en mémoire des objets suppose une mise à jour des adresses (dans les racines et dans les objets). Ceci peut être facilement réalisé en maintenant une table de correspondance nouvelle/ancienne adresse ou en utilisant la table d'indirection utilisée pour implémenter les références si elle existe. Outre l'absence de la deuxième phase coûteuse du *Mark and Sweep*, cette approche permet d'éviter la fragmentation du tas (les blocs mémoires sont directement compactés dans le tas inactif). En particulier, l'allocation devient beaucoup plus efficace, puisque le calcul de l'adresse du prochain bloc mémoire libre est immédiat.



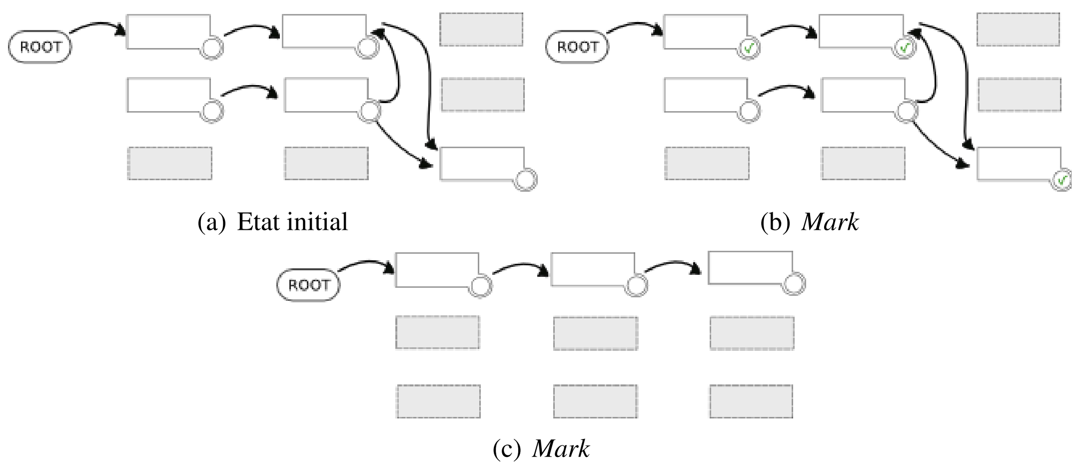
(a) Etat initial

(b) Etat final

FIGURE 4 – *Copying*

- \+ Technique simple,
- \+ Bonnes performances,
- \+ Compactage du tas, pas de fragmentation et augmentation des performances de l'allocation,
- \- Doublement de la taille du tas.

Mark and Compact L'algorithme *Mark and Compact* représente un compromis entre le *Mark and Sweep* et le *Copying*. La première phase (*Mark*) est identique à celle du *Mark and Sweep*. La seconde phase compacte les blocs mémoire marqués au début du tas.



(a) Etat initial (b) *Mark*

(c) *Mark*

FIGURE 5 – *Mark and Compact*

- \+ Compactage du tas, pas de fragmentation ;
- \+ Augmentation des performances de l'allocation ;

- Parcours intégral du tas.

Comme dans l'algorithme *Copying*, les références doivent être mises à jour.

4.6.2.3 Optimisations

Incremental garbage collector La suspension de l'application pendant toute la durée de l'exécution du *garbage collector* peut être pénalisante pour des applications réactives. Pour limiter l'impact de cette suspension, l'exécution d'un *garbage collector* peut être incrémentale. Dans ce cas, chaque phase de son exécution est réalisée en plusieurs pas, entrelacés avec l'exécution du programme. L'implémentation d'un *garbage collector* incrémental demande certaines précautions pour s'assurer que l'exécution du programme entre deux pas n'invalide pas son analyse des objets accessibles. Par exemple, si deux variables `x` et `y` sont respectivement initialisées à `null` et à l'adresse d'un objet quelconque et que leurs contenus sont ensuite inversés. Si la phase du marquage (en considérant l'algorithme *Mark and Sweep*) est interrompue juste avant l'inversion (après avoir uniquement exploré `x`), l'adresse de l'objet contenu dans `y` n'a donc pas encore été marquée. Supposons maintenant que l'exécution du *garbage collector* reprenne après l'inversion. Il lui reste à explorer `y`, mais l'adresse n'est toujours pas marquée puisque qu'elle n'est plus dans `y`. L'adresse n'est donc jamais marquée et pourtant elle est bien accessible. Ce problème peut être résolu en identifiant les modifications réalisées pendant que le *garbage collector* est suspendu (voir le *garbage collector mostly concurrent* ci-dessous).

garbage collector mostly concurrent La notion de *garbage collector mostly concurrent* a été proposée par Boehm. Il s'agit d'un *garbage collector*, basé sur l'agorithme *Mark and Sweep*, dont une partie importante de l'exécution peut être réalisée en parallèle avec celle du programme. Plus précisément, la phase de marquage des éléments du tas et la phase de nettoyage sont réalisées en parallèle de l'exécution du programme. Au cours d'une première phase, durant laquelle le programme est suspendu, l'algorithme réalise un marquage des racines. Dans un second temps, en parallèle de l'exécution du programme, le marquage du reste du tas est effectué. L'exécution parallèle de cette phase fait que l'information calculée par le *garbage collector* peut être invalidée. Pour cela, les modifications du tas faites par le programme au cours de cette phase sont enregistrées (drapeau « *dirty* » sur les références). Le programme est suspendu et une nouvelle phase de marquage démarre, cette fois à partir des références marquées *dirty*. Il est possible de réaliser cette phase en parallèle avec le programme, mais les modifications du tas effectuées par le programme obligent à recommencer jusqu'à ce que le programme soit suspendu pour obtenir un résultat stable. La phase *Sweep* de l'algorithme peut être réalisée en parallèle de l'exécution du programme.

Generational garbage collector Les *garbage collectors* générationnels s'appuient sur l'observation que toutes les références n'ont pas la même durée de vie. La plupart des objets ont une durée de vie très courte et seul un petit nombre ont une durée de vie très longue. Dans un

garbage collector générationnel, le tas est divisé en plusieurs parties, chacune correspondant intuitivement à l'âge des objets qu'elle contient. Le cas de figure le plus simple comporte deux générations : jeune et ancienne. Dans ce cas, un nouvel objet est alloué dans la jeune génération. Lorsqu'une phase de *garbage collection* est nécessaire, on essaie d'abord de la réaliser sur la jeune génération (*minor collection*). Un objet qui survit à un nombre fixé de *minor collections* est promu à l'ancienne génération. Si cette étape n'est pas suffisante, on procède à une phase de *garbage collection* sur l'ancienne génération (*major collection*). Dans la plupart des cas, la première phase est suffisante, ce qui permet un gain de temps important, puisque des objets dont la durée de vie est longue seront testés moins souvent. L'implémentation d'un *garbage collector* générationnel nécessite de mettre en place des mécanismes de gestion des pointeurs inter-générations sans quoi une référence de la jeune génération, accessible seulement depuis une référence de l'ancienne génération (elle-même accessible depuis la racine) pourrait être libérée à tort.

Parallel garbage collector Un *garbage collector* parallèle, à ne pas confondre avec concurrent, est un *garbage collector* dont l'algorithme est parallèle. Il n'existe pas de spécification plus précise de ce type de *garbage collector*, le cas particulier de celui de HotSpot est présenté dans [5].

4.6.3 Conclusion

L'impact direct d'un *garbage collector* sur la sécurité est faible, les risques reposent avant tout sur la complexité de l'algorithme utilisé et sur son implémentation. Le *garbage collector* de HotSpot, par exemple, atteint un degré de complexité important, car il combine plusieurs approches dont il faut maitriser les interactions. En particulier, l'implémentation d'un *garbage collector* parallèle efficace est une tâche très difficile dans laquelle beaucoup d'erreurs peuvent se glisser. La question des *garbage collectors* conservatifs ne pose un problème de sécurité que si l'on se repose sur la gestion automatique de la mémoire pour assurer l'effacement de données sécurisées, ce qui, de manière générale, est une mauvaise pratique.

4.7 Intégration avec les mécanismes de l'OS

Le cloisonnement applicatif entre instances de la JVM est réalisé de manière transparente par le système d'exploitation. Ce cloisonnement permet de protéger les accès mémoire entre instances, même si certains segments mémoire sont tout de même partagés (par exemple, le code binaire de la JVM, les bibliothèques partagées ou les classes système). En effet, le contrôle d'accès géré par l'OS autorise à avoir un segment, en lecture seule, partagé entre différentes instances (ou processus). Cependant, dès qu'une tentative d'écriture est effectuée dans un segment mémoire, une copie privée de ce dernier est réalisée dans le processus courant.

Les systèmes d'exploitation récents intègrent un mécanisme permettant de rendre aléatoire les adresses mémoire des segments de données. Ce mécanisme rend plus difficile la localisation, au sein de l'espace d'adressage mémoire d'un processus, des éléments utiles pour mener une attaque. Cette protection ne gêne en rien le fonctionnement d'une application standard, telle qu'une JVM.

Une autre protection, le bit NX, implémentée dans les processeurs récents, permet d'interdire l'exécution de code au sein d'une page mémoire marquée comme non-exécutable. Cette protection permet de protéger une application notamment face aux attaques de type débordement de pile. Une telle protection n'est pas contraignante pour le fonctionnement de la JVM, quel que soit le mode d'exécution implémenté (interprétation ou *JIT*).

Les protections mémoire avancées, comme PaX, offrent un niveau de robustesse intéressant vis-à-vis de nombreuses formes d'attaques. Cependant, ces protections ne sont pas toujours compatibles avec le fonctionnement des applications qu'elles protègent. C'est notamment le cas des applications qui réalisent des manipulations non classiques telles que la modification, pendant l'exécution, du code exécutable. Les implémentations de la JVM effectuant de la compilation « à la volée » ou des formes optimisées d'interprétation (mode *inline* ou mode *template*), sont typiquement concernées par cette incompatibilité. Il est donc nécessaire d'avoir connaissance du fonctionnement interne de la JVM utilisée lorsque des protections mémoire avancées doivent être mises en place.

Au final, il est possible de tirer partie des protections mises en œuvre par le système d'exploitation. Les mécanismes de protection mémoire sont pour la plupart compatibles avec le fonctionnement de la JVM. Seule la protection mémoire avancée, de type PaX, peut empêcher, sous certaines circonstances (mode *JIT* et/ou mode interprété *inline* ou *template*) le fonctionnement de la JVM.

5 MODÈLE D'EXÉCUTION NATIF

5.1 Introduction

Cette section traite du mode d'exécution natif. Il est intéressant de préciser ce que nous entendons par « mode d'exécution natif », dans le contexte des modèles d'exécution Java. Il s'agit de la stratégie d'implémentation de la machine virtuelle Java qui repose principalement sur la compilation statique (c'est-à-dire avant exécution) de l'application. Cette stratégie repose sur l'utilisation d'un compilateur *AOT* (*Ahead-of-Time*) qui est un outil de conversion d'un *bytecode* Java en code natif pour le processeur cible. Le compilateur *AOT* est généralement utilisé pour fournir un exécutable autonome au format adapté à la machine cible. Ce format sera en général le même format que celui utilisé pour les programmes C et C++ de la plate-forme : sous Linux, il s'agit du format ELF.

L'exécution d'un programme en Java converti à ce format ne nécessite, en théorie, plus d'interprétation [6].

Les avantages de cette approche sont les suivants :

- la vitesse d'exécution est proche de celle d'un programme implémenté en C/C++ ;
- il n'est plus nécessaire d'installer un émulateur sur la plate-forme d'exécution native ;
- l'exécution ne souffre plus du temps de latence lié au chargement de l'émulateur.

Par contre, les inconvénients sont :

- ce mode d'exécution reste marginal : il est très peu supporté par les outils, aussi bien *open source* que commerciaux ;
- une fois compilé au format natif, un programme Java est **beaucoup** plus volumineux que son équivalent en *bytecode*. En effet, l'implémentation des vérifications dynamiques, normalement faites par la JVM dans le mode d'exécution par émulateur, est ici répétée dans le code machine du programme pour chaque vérification dynamique, ce qui engendre un surcoût important en taille de code généré.

5.2 Historique

L'approche native a connu son heure de gloire quand le mode interprété était encore d'une lenteur affligeante, avant l'introduction des techniques de *JIT*.

6. L'étape de chargement de classes décrite dans la spécification de la JVM étant dynamique, il se peut que certaines classes ne puissent être déterminées statiquement et nécessitent d'être chargées lors de l'exécution du programme. Un interpréteur doit donc dans ce cas être embarqué dans l'exécutable pour exécuter les méthodes de ces classes.

Les principales solutions de l'époque étaient :

- GCJ, `http://gcc.gnu.org/java/`
- JET, `http://www.excelsior-usa.com`
- JOVE, `http://www.instantiations.com/home.htm`
- NaturalBridge BulletTrain compiler `http://www.naturalbridge.com/`
- Manta Fast Parallel Java `http://www.cs.vu.nl/manta/`
- The Timber Compiler `http://www.pds.twi.tudelft.nl/timber/`
- Diab-SDS's fastJ compiler
- Visual Café `http://www.webgain.com/`
- Visual Age `http://www-4.ibm.com/software/ad/`

(Source `http://www.bearcave.com/software/java/comp_java.html`)

De ces solutions, seules GCJ et JET existent encore aujourd'hui [7], et GCJ est le seul à supporter des environnements non x86. Les autres ont disparu, ou, pour les projets *open source* (Manta Fast Parallel Java, The Timber Compiler), aucune nouvelle version n'est apparue depuis au moins 5 ans.

5.3 Implémentations

Les deux implémentations encore en course se distinguent par les éléments suivants :

- L'implémentation du projet GNU est avant tout motivée par un aspect idéaliste (fournir une implémentation de Java sous licence GPL). En second plan interviennent les considérations techniques, telles que la plus grande facilité d'intégrer Java dans l'environnement GNU en passant par le mode natif plutôt que par l'approche utilisant un émulateur. Le projet GNU met également en avant le fait que la compilation native rend le code source Java plus portable, car GCJ supporte plus de plates-formes cibles que ne le supportent les émulateurs du marché.
- L'implémentation de Excelsior JET met en avant la réduction de l'empreinte mémoire (à la fois en mémoire de masse et en mémoire vive), un temps de chargement réduit ainsi qu'une optimisation poussée disponible dès le chargement du programme. JET existe pour Windows et Linux.

7. Certaines des pages citées ci-dessus ne font plus directement référence aux compilateurs sur leur page principale, mais des informations sont encore disponibles sur ces sites.

5.4 Architecture canonique

On décrit ici une architecture de référence d'un environnement d'exécution natif. On peut en effet parler ici d'implémentation de référence, car il s'agit de celle respectant au mieux la spécification de la JVM. Elle est supportée aussi bien par JET que GCJ.

Dans cette architecture, la compilation est réalisée en deux étapes :

- une première étape de compilation des sources Java vers une forme *bytecode*, en utilisant un compilateur Java dit « classique » ;
- une deuxième étape de compilation du *bytecode* vers le code natif. Ce rôle est dévolu au compilateur dit « natif ».

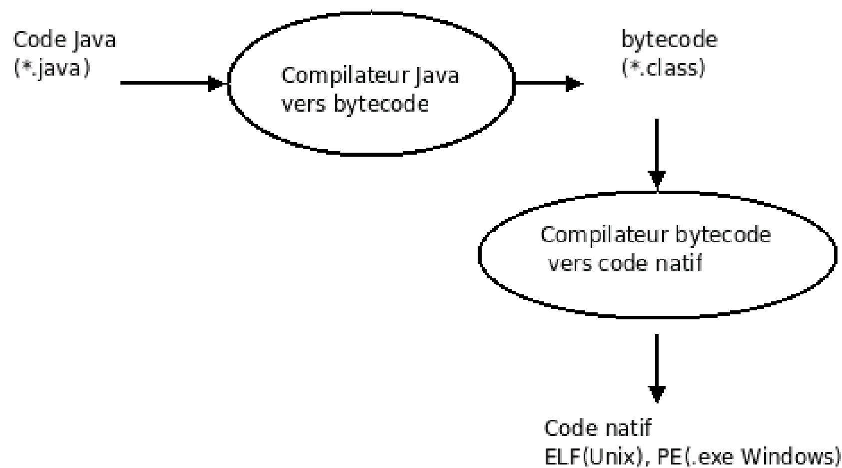


FIGURE 6 – Architecture canonique de compilation native

Cette architecture possède les avantages suivants :

- elle est plus modulaire. En pratique cette implémentation permet d'utiliser un compilateur Java classique pour parser le source Java ;
- elle préserve le format *bytecode* comme produit intermédiaire de compilation, ce qui la rapproche d'une implémentation classique par émulateur (cf. section 5.5 pour les avantages de ce point).

5.5 Architecture alternative

Une autre solution architecturale permettrait de passer directement du code Java au code natif, sans passer par une représentation intermédiaire de type *bytecode*. Cette solution, bien que techniquement possible, pose la question de la transposition des vérifications statiques effectuées normalement sur le *bytecode* vers des vérifications effectuées durant le processus de compilation (donc à partir d'une représentation de plus haut niveau). En effet, la spécification

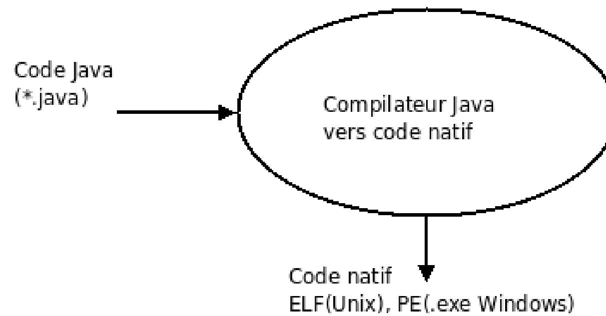


FIGURE 7 – Architecture alternative de compilation native

de la JVM précise les vérifications à effectuer au niveau du *bytecode* : il existe donc une spécification assez précise du vérificateur de *bytecode*. Comment réaliser la transposition de ces vérifications dans une architecture où le *bytecode* n'existe plus en tant que tel ?

Nous pouvons distinguer trois catégories de vérifications statiques :

- les vérifications qui n'ont plus lieu d'être effectuées, car devenues caduques. Ces vérifications sont les vérifications de première passe telles que définies par la spécification de la JVM [15]. Il s'agit de vérifier que les fichiers `.class` sont bien formés ;
- les vérifications qui sont redondantes avec la sémantique du langage Java, comme par exemple le fait de vérifier qu'une classe `B` n'hérite pas d'une classe `A` déclarée `final`, ou que toute classe possède une classe parente, à l'exception de la classe `java.lang.Object`. Il s'agit des vérifications de deuxième passe telles que définies par la spécification de la JVM [15] ;
- les vérifications de passe 3 ou 4 telles que définies par la spécification de la JVM [15], qui nécessitent une compréhension plus poussée du code (analyse de flot de données), et qui s'appliqueront vraisemblablement sur une représentation interne du code compilé spécifique à chaque compilateur.

Dans une première approche, on peut affirmer que cette transposition n'est pas triviale, et que l'implémenter correctement est une opération beaucoup moins évidente qu'implémenter un simple vérificateur de *bytecode*. Il semble aussi plus difficile d'auditer l'implémentation de l'équivalent du vérificateur de *bytecode* dans cette architecture que dans l'architecture canonique.

Se pose également le problème de la « distance » entre l'étape de vérification et celle d'exécution de l'application. Concrètement, il est nécessaire de garantir l'intégrité du code natif une fois que celui-ci est compilé afin de se prémunir des éventuelles modifications effectuées par un attaquant.

Il existe un projet *open source* se rapprochant de cette architecture alternative, le projet JC [8]. Toutefois, ce projet ne représente pas tout à fait cette architecture, car :

8. `http://jcvm.sourceforge.net`

- il passe par une étape intermédiaire de transformation du code Java en C, avant compilation par le compilateur C GCC ;
- il est plus riche en fonctionnalités (support d'un interpréteur en sus de la compilation native par exemple).

5.6 Problématique de stockage et d'intégrité du code natif

Les implémentations *AOT* se démarquent des émulateurs (utilisant éventuellement un compilateur *JIT*) par une approche agressive de l'optimisation lors de la compilation (c'est-à-dire avant exécution ou lors du lancement de l'application).

Ces optimisations agressives sont en revanche très coûteuses en temps *CPU*. Elles sont similaires aux méthodes d'optimisation des compilateurs C ou C++, où un temps de compilation de plusieurs minutes, voire plusieurs heures pour des applications conséquentes, est considéré comme « acceptable ».

Par conséquent, pour les implémentations *AOT*, une fois qu'une classe (ou un JAR) est convertie au format natif, cette dernière est en général stockée sur disque dans une base de données, afin de pouvoir être rechargée plus tard.

Ceci a un effet de bord intéressant d'un point de vue sécurité. En effet, une JVM traditionnelle se protège des attaques par injection de *bytecode* non conforme en passant le vérificateur de *bytecode* sur le *bytecode* avant de l'exécuter. Dans notre cas, le vérificateur de *bytecode* sera forcément passé **avant** la compilation *AOT*. Ce qui veut dire que l'intégrité du code machine stocké dans la base de données doit être assurée. Ceci est une contrainte supplémentaire par rapport au modèle d'exécution purement interprété, ou à un *JIT* qui ne conserve pas le code machine généré d'une exécution sur l'autre.

5.7 Gestion des *threads*

La problématique de gestion des *threads* après compilation native est globalement similaire à celle rencontrée dans un environnement d'exécution par émulateur. On pourra se reporter à la section 4.5 pour plus de détails.

5.8 Gestion de la mémoire

La problématique de la gestion mémoire est similaire à celle du modèle d'exécution par émulateur. On pourra se reporter à la section 4.6 pour plus de détails.

5.9 Gestion du chargement de classes

La spécification de la JVM précise que le chargement de classes est réalisé dynamiquement, mais cette étape peut dans la plupart des cas être réalisée statiquement, avant exécution. Toutefois, certaines fonctionnalités de Java (par exemple, la possibilité pour le développeur de spécifier son propre chargeur de classes, les mécanismes d'exécution de code mobile tels que les applets Java) nécessitent explicitement que le chargement de certaines classes soit réalisé lors de l'exécution, le type de certaines classes ne pouvant être déterminé statiquement.

Dans le cas d'un modèle utilisant uniquement la compilation *AOT*, l'ensemble des classes doit être connu et compilé avant l'exécution de l'application. Il y a alors un choix architectural à faire :

- soit le modèle ne supporte pas le chargement dynamique ce qui constitue une déviation par rapport au modèle d'exécution JVM (le chargement dynamique de *bytecode* à l'exécution n'est plus supporté et la distribution des applications n'est plus effectuée sous forme de fichier `class`).
- soit un interpréteur Java (ou un *JIT*) est intégré ou lié à l'exécutable natif. Le modèle se rapproche alors de la stratégie par utilisation d'un émulateur.

On voit ici qu'il n'y a pas de solution « idéale » pour la problématique du chargement de classes. Soit on inclut dans le binaire de l'application un interpréteur, ce qui est une solution relativement lourde et complexe, soit on supprime une fonctionnalité phare de la plate-forme Java.

5.10 Bibliothèques de support à l'exécution

L'exécutable résultant du processus de compilation native se base sur un *runtime* d'exécution de l'environnement Java, implémenté sous Linux par des bibliothèques partagées (fichiers `.so`). Ce *runtime* comprend les éléments suivants :

- L'ensemble des classes de la bibliothèque standard pré-compilées sous une forme directement exécutable par le processeur ;
- des fonctions de support des mécanimes de la JVM (*garbage collector*, gestion des *threads*, etc) ;

– éventuellement un interpréteur de *bytecode* Java.

Par exemple, un programme une fois traduit en code natif par GCJ est lié à la bibliothèque `libgcj.so`. La taille de cette bibliothèque dynamique est de plus de 30 Mo (GCJ 4.3.3, compilation 32 bits).

Par conséquent, le modèle d'exécution natif ne permet pas d'avoir un réel exécutable autonome, puisqu'il est accompagné d'un *runtime* qui peut être de taille conséquente.

5.11 Implémentation native de la bibliothèque standard

Au delà de l'utilisation de compilateurs AOT et JIT pour passer en mode natif, il est intéressant de noter que l'amélioration des performances d'une plate-forme d'exécution Java repose en partie sur la qualité d'implémentation de la bibliothèque standard (seules les « couches hautes » de la bibliothèque, c'est-à-dire l'API, étant standardisées). Une stratégie d'optimisation consiste alors à recourir massivement à l'interface JNI et à implémenter en C ou C++ les « couches basses » de la bibliothèque. Cette utilisation du code natif présente des avantages (meilleures performances, meilleure intégration avec le système, notamment en ce qui concerne les primitives graphiques). Elle constitue cependant une déviation importante du modèle, puisque les propriétés du langage Java ne sont pas assurées pour ce code natif. Celui-ci peut notamment comporter des vulnérabilités que l'utilisation de Java est censée éviter (typiquement, les débordements de tampon ou *buffer overflow*). Du point de vue de la sécurité, l'utilisation de JNI dans l'implémentation de la bibliothèque standard devrait être limitée à l'interface avec les couches basses de l'OS (typiquement, les appels système) afin de réduire la portion de code natif « non-Java » utilisé par une application.

5.12 Conclusion

Il est intéressant de noter que le mur de séparation entre mode d'exécution natif et l'utilisation d'un émulateur s'est considérablement effrité avec le temps. En effet, les JVM performantes incluent toutes un *JIT*, qui n'est rien d'autre qu'une compilation native, mais effectuée durant l'exécution d'un programme Java. De même, une approche qui privilégierait une compilation complète avant exécution du programme Java (approche *AOT*), qu'on appelle ici compilation native, doit inclure un interpréteur ou un *JIT* afin d'assurer la compatibilité avec les applications Java qui utilisent le chargement de classes dynamique.

Le modèle d'exécution natif (compilation statique ou *AOT*) est caractérisé par un manque flagrant d'étude approfondie sur son mode de fonctionnement détaillé. Cette stratégie d'implémentation souffre d'une distance importante entre l'implémentation et la spécification de la JVM. Au delà de la conformité à la spécification, se pose le problème de la vérification de

cette conformité (notamment en ce qui concerne les aspects liés à la sécurité). Le besoin industriel pour ce type d'approche n'est aujourd'hui plus évident, les performances des émulateurs s'étant considérablement améliorées par rapport aux premières versions de Java, surtout depuis l'arrivée d'une forme de compilation native (*JIT*). Il est intéressant de noter qu'au delà de l'utilisation des compilateurs *AOT* et *JIT*, l'amélioration des performances d'une plate-forme d'exécution Java repose en partie sur la qualité d'implémentation de la bibliothèque standard, seules les « couches hautes » de la bibliothèque (c'est-à-dire l'API) étant standardisées.

Une des raisons qui pourraient pousser à vouloir utiliser le mode d'exécution natif est la possibilité d'aller beaucoup plus loin dans l'offuscation de code que ce qu'il est possible de faire en restant dans le monde Java. Il ne faut cependant pas oublier que le passage au mode natif n'est pas anodin, et qu'il peut y avoir un impact sur les propriétés de sécurité de l'application ainsi transformée. On pensera en particulier au code machine de l'application transformée qui peut avoir éventuellement besoin d'être protégé en intégrité, si le mode de transport de celui-ci permet à un attaquant de l'altérer.

Les implémentations GNU (GCJ) et Excelsior (JET) sont les seules survivantes, et aucune n'a été véritablement conçue dans une optique du développement d'applications de sécurité. La raison d'être de GCJ était de fournir une implémentation entièrement libre du langage Java. Excelsior quant à elle vise plutôt l'aspect performance.

GCJ et Excelsior sont toutes les deux à jour vis-à-vis des derniers standards Java. Cependant, on peut se poser la question de leur pérennité à long terme. L'intérêt pour GCJ est retombé depuis qu'OpenJDK (issu de la libération du code source de l'implémentation Sun) est apparu et que Redhat s'est maintenant détourné de GCJ au profit d'OpenJDK.

Du fait des progrès continus réalisés par le *JIT*, on peut également se poser la question de l'intérêt de l'approche d'Excelsior, dans la mesure où les performances, qui hier étaient clairement en faveur d'Excelsior, sont aujourd'hui à peu près équivalentes à celles d'une JVM avec un *JIT* performant.

6 MODÈLES D'EXÉCUTION PAR PROCESSEUR SPÉCIALISÉ

6.1 Introduction au modèle d'exécution

Il existe deux approches *Java processor* différentes pour améliorer l'exécution du *bytecode* Java de façon matérielle :

1. **l'approche par coprocesseur Java**. Cette première méthode consiste à utiliser un « coprocesseur » Java conjointement au processeur classique. Ce coprocesseur peut alors traduire le *bytecode* Java en une suite d'instructions RISC compréhensibles par l'autre processeur ou alors l'exécuter directement.
2. **l'approche par remplacement du processeur principal**. Cette autre méthode utilise des processeurs Java qui viennent remplacer le processeur classique. Ceci implique que toutes les applications qui s'exécutent doivent être écrites dans un langage compilable vers le *bytecode* Java. Ce type de processeur est donc particulièrement adapté au monde de l'embarqué.

L'approche *Java processor* a été une approche envisagée pour permettre l'exécution de *bytecode* Java sur des plates-formes embarquées disposant de très peu de ressource mémoire et *CPU*. À cette époque, l'utilisation d'un émulateur n'était pas envisageable sur ces plates-formes, du fait de la pénalité en temps d'exécution et en occupation mémoire introduite par l'émulateur.

Ce modèle s'appuie sur une implémentation (souvent partielle) de la JVM sous forme matérielle. Le jeu d'instructions de la JVM (les instructions du *bytecode*) est généralement implémenté seulement partiellement dans le silicium. Le but est de rendre le processeur capable d'exécuter directement les instructions Java les plus simples (calculs numériques, opérations de `load` et `store`, tests de branchement), et qui sont aussi celles rencontrées le plus souvent dans les fichiers `class`. Le *bytecode* ne comportant pas d'instructions permettant d'accéder aux ressources natives (par exemple, pour la gestion des entrées/sorties), ce type de processeur comprend également d'autres instructions permettant d'effectuer ces accès ou utilise le processeur « générique » dans le cas d'un coprocesseur. De plus, certains services de la JVM (par exemple, la gestion de la mémoire ou le chargement de classes) étant difficilement implémentables sous forme matérielle, ce modèle d'exécution s'appuie également sur une implémentation logicielle (généralement désignée sous le terme de JVM) qui fournit ces services et s'appuie sur le processeur dédié pour l'exécution du *bytecode*.

Pour accéder à une implémentation logicielle, les processeurs utilisent généralement le mécanisme des « *traps* » ou interruptions logicielles. Le « *trap* logiciel » est une interruption classique, à la différence qu'elle n'est pas prise en compte par le processeur à partir de signaux externes provenant d'autres composants, mais induite par l'exécution d'une instruction spécifique du langage machine. Après l'exécution de la procedure principale de gestion de l'interruption qui contient le code permettant d'émuler la fonction à exécuter, le processeur pourra reprendre son exécution normale.

6.2 Périmètre de l'étude

La suite de l'étude présente une liste de solutions proposées par des fournisseurs. Il est important de noter que les détails d'implémentation et d'architecture de chaque processeur Java ont été ici laissés de côté. En effet, on se heurte dans ce domaine à un problème d'accessibilité de l'information technique. À la différence des modèles d'exécution par émulateur (JVM) ou alors de JavaCard, les fonctionnements internes des processeurs Java sont peu documentés. Les seules documentations mises à disposition par les fondeurs de Java Processors sont plutôt des informations macroscopiques sous forme de schémas d'architecture. Ce type d'information est plutôt destiné à des donneurs d'ordre des fabricants de périphériques mobiles.

Par conséquent, l'étude suivante constitue plus une liste des produits existants ou ayant existé, en précisant leurs caractéristiques principales et leurs limites, qu'une étude poussée du mode de fonctionnement de chaque solution.

6.3 Solutions étudiées

Dans la suite de ce document, les solutions suivantes seront étudiées.

Co-processeurs :
- Jazelle [9].

Processeurs :
- IM1101(Cjip) [10].
- picoJava [11] ;
- aJile [12] ;
- JOP [13] ;
- IM3000 [14].

Les informations accessibles sur les différentes architectures possibles, ou sur les différents processeurs, ne permettent pas de cerner précisément quels sont les gains/pertes au niveau sécurité par rapport au modèle d'exécution par émulateur.

9. http://www.arm.com/products/multimedia/java/jazelle.html
10. http://www.imsystech.com/
11. http://www.sun.com/processors/technologies.html
12. http://www.ajile.com/
13. http://www.jopdesign.com/
14. http://www.imsystech.com/

6.4 picoJava

Le processeur Java picoJava de Sun est le plus souvent cité dans les documents de recherche. Il s'agit en réalité d'une spécification utilisée comme référence pour les nouveaux processeurs Java et comme base de recherche pour l'amélioration de processeurs existants.

6.4.1 Historique

La première version a été introduite en 1997. Le marché visé est celui des systèmes embarqués, en offrant un « pur » processeur Java. Une seconde version, picoJava-II, a ensuite vu le jour en 1999.

6.4.2 Implémentation

Etant donné qu'il s'agit d'une spécification, ce n'est qu'une base vers la création réelle de processeurs. Sun n'a jamais lui-même produit (de manière industrielle) de processeurs respectant picoJava. Bien que des licences aient été fournies à Fujitsu, IBM, LG Semicon et NEC, ces sociétés n'ont pas non plus produit de processeurs picoJava.

6.4.3 Version du langage Java supporté

Pas d'informations disponibles

6.4.4 Evolution

Toutes les sources nécessaires à la réalisation du processeur sont désormais disponibles grâce au Sun Community Source Licensing. Mis à part le projet Harvey réalisé par Wolfgang Puffitsch, il n'existe pas de processeur Java qui revendique clairement être un processeur picoJava. Harvey est une implémentation d'un processeur picoJava dans un FPGA. Il est disponible sous licence LGPL (dernière mise à jour le 3 janvier 2008).

6.5 aJile

6.5.1 Historique

Courant 1997, Rockwell Collins a annoncé avoir réalisé le premier processeur Java, le JEM1. Ce processeur, qui n'était pas basé sur le modèle proposé par Sun, fut créé pour une utilisation en interne. Ajile Systems commercialise ensuite l'aJ-100, basé sur le processeur JEM2 et créé en 1999 par des ingénieurs issus de Rockwell Collins.

6.5.2 Implémentation

Ce processeur autorise l'exécution de plusieurs machines virtuelles simultanément. Chaque machine virtuelle possède alors son propre espace mémoire, ses propres *threads*, tout en ayant un niveau d'isolation garanti par l'implémentation du modèle « bac à sable » (ou *sandbox*) aussi utilisé par les plates-formes J2SE.

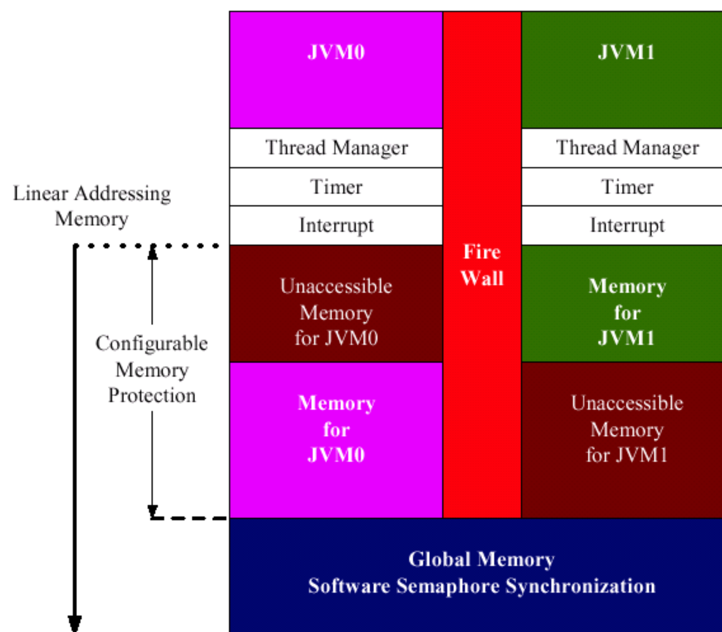


FIGURE 8 – Gestion de la mémoire avec aJile

6.5.3 Version du langage Java supporté

L'environnement d'exécution Java temps réel d'aJile est implémenté suivant les recommandations CLDC 1.1 (*Connected Limited Device Configuration*) et CDC 1.1 (*Connected Device Configuration*). Le processeur aJ-100 est donc un processeur compatible avec la plate-forme J2ME.

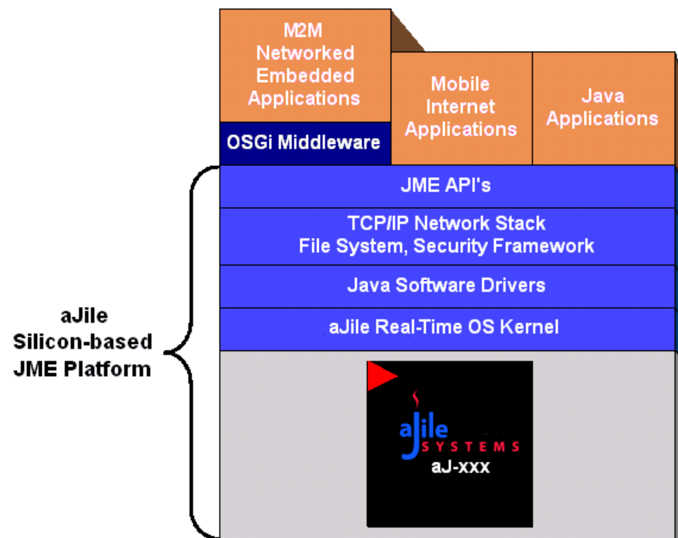


FIGURE 9 – Plate-forme J2ME d'aJile

6.5.4 Evolution

aJile Systems propose une nouvelle génération de son processeur Java, du nom d'aJ-102. Selon la compagnie, ce nouveau venu traite les instructions Java trois fois plus rapidement que son prédécesseur. Cette puce tout-en-un intègre diverses unités de traitement (nombres flottants et chiffrement AES), de multiples contrôleurs (mémoire, USB 2.0, Ethernet à 10/100 Mb/s, écran LCD, etc.) et un ensemble complet d'entrées/sorties. Le tout est livré avec un système d'exploitation temps-réel, entièrement écrit en Java.

L'aJ-102 est disponible depuis avril 2009 au tarif de 16 dollars HT. La compagnie prosera également l'aJ-200 : cette puce sera cadencée à 180 MHz et proposera une unité multimédia évoluée capable de supporter les formats JPEG, MPEG et H.263.

6.6 Imsys Technologies AB

6.6.1 Historique

La société suédoise Imsys Technologies AB produit deux séries de processeurs capables d'exécuter du *bytecode* Java. La série IM1000, qui est produite depuis 1999 sous le modèle CJip IM-1101 [7] (anciennement GP1000), et la série IM3000 composée des modèles IM-3221 et IM-3910 [8]. Il existe également deux autres processeurs non-compatibles Java dans la série IM3000, le IM-3220 et le IM-3240.

6.6.2 Implémentation

Le processeur Java IM-1101 embarque une machine virtuelle Java implémentée en microcode alors que les processeurs Java de la série IM3000 possèdent une machine virtuelle basée sur KVM (*Kilobyte Virtual Machine*) qui est une version plus légère de la JVM de Sun.

Le jeu d'instruction des processeurs de Imsys se limite à 85% du jeu d'instructions standard du langage Java contre 99% pour aJile et son processeur Aj-100. En effet, les instructions les plus complexes et les plus rares ne sont pas implémentées, mais exécutées de façon logicielle [16].

Ces processeurs supportent le langage Java, mais également le C/C++ et le langage assembleur.

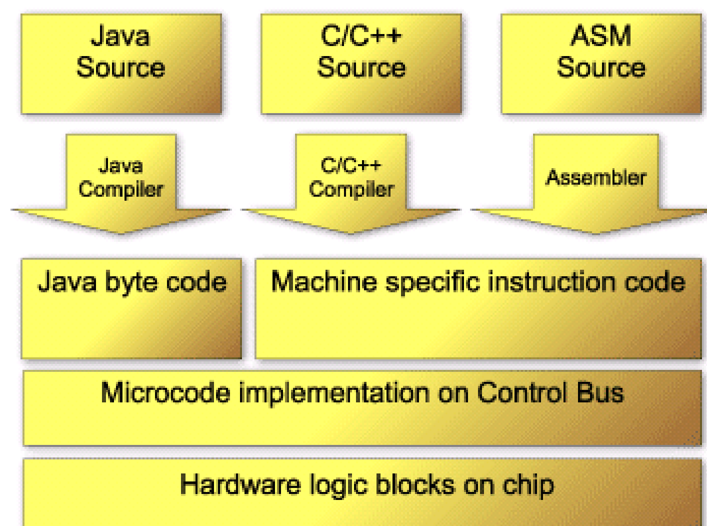


FIGURE 10 – Langages supportés par et architecture du processeur IM-1101

Contrairement à aJile, les processeurs de la série IM3000 ne peuvent avoir qu'une seule machine virtuelle active simultanément. Néanmoins, plusieurs applications Java peuvent être exécutées en même temps et partager alors le même « tas ».

Sur cette famille de processeurs, l'exécution du *Garbage Collector*, qui se déroule dans un *thread* distinct, peut être réalisée automatiquement lorsque le « tas » est plein, ou de façon explicite depuis un programme Java ou C.

6.6.3 Version du langage Java supporté

Processeurs de la série IM1000 : L'association du processeur IM-1101 avec l'OS temps-réel Moose, également développé par Imsys, permet le support de la plate-forme J2ME avec le *Connected Limited Device Configuration* (CLDC). Le seul profil J2ME implémenté est le *Mobile Information Device Profile*, désigné par l'acronyme *MIDP*, qui est un profil utilisé par certains téléphones mobiles.

Processeurs de la série IM3000 : L'API de base de ces processeurs est le *Connected Limited Device Configuration* (CLDC) version 1.0. Elle contient également un sous-ensemble du JDK 1.1.8, le package `javax.comm`, ainsi qu'un ensemble de classes spécifiques à Imsys permettant aux applications d'accéder aux ressources système.

6.6.4 Evolution

Les processeurs réalisés par Imsys sont toujours disponibles, mais il n'y a pas d'information disponible sur les futures réalisations de cette société.

6.7 JOP

6.7.1 Historique

Le processeur JOP est en réalité un « cœur logiciel » basé sur l'idée qu'une implémentation native de la totalité du *bytecode* Java n'est pas une approche utile.

- 2000 : JOP1 est la première version de ce processeur et est écrite en AHDL.
- Avril 2001 : une deuxième version est cette fois réalisée en VHDL.

– Juin 2001 : création de JOP3, qui est la version actuelle et est généralement désignée par le terme JOP.

Le projet JOP (*Java Optimized Processor*) a été l'objet de la thèse réalisée par Martin Schoeberl en 2004 à l'université de Vienne (voir [11, 12] pour plus de détails). Depuis le 24 février 2008, ce projet *open source* est passé sous la licence GNU General Public Licence version 3. Un des principaux objectifs de ce projet est de pouvoir prévoir la durée d'exécution du *bytecode* Java pour des systèmes temps-réel.

6.7.2 Implémentation

Le processeur JOP possède son propre jeu d'instructions. Ce dernier, appelé « microcode », est un jeu d'instructions réduit permettant de traduire une instruction *bytecode* en une ou plusieurs instructions microcodes. Contrairement aux autres processeurs Java, JOP n'utilise pas le mécanisme des « *traps* » (défini dans le paragraphe 6.1) pour émuler le *bytecode* Java qui n'est pas directement implémenté en microcode. Le processeur passe donc par une table de correspondances qui contient, pour chaque instruction *bytecode*, l'adresse de la suite d'instructions microcode à exécuter. Pour toutes les instructions *bytecode* n'ayant pas de correspondance en microcode, l'adresse inscrite dans la table pointe sur une séquence d'instructions qui invoque une méthode statique. Cette méthode statique, qui est du *bytecode* Java, est à son tour traduite en microcode à l'aide de la table de correspondance.

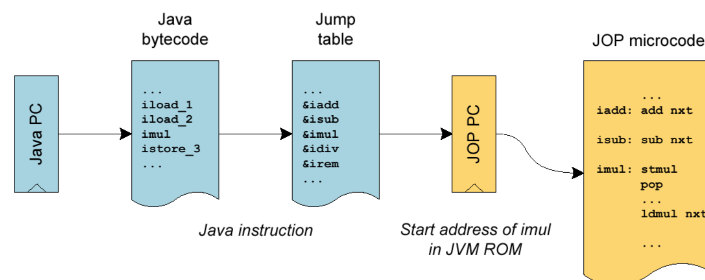


FIGURE 11 – Passage du programme Java au microcode JOP

Ce processeur, de part sa petite taille, peut être implanté dans un FPGA standard de type Altera (ACEX 1K50, Cyclone) et Xilinx (Spartan II et Spartan-3). L'implantation du programme Java est réalisée à l'aide de l'outil JOPizer. Ce dernier réalise la vérification du *bytecode*, l'édition des liens, puis la transformation en un fichier `.jop`. C'est ce fichier qui sera ensuite implanté sur le FPGA cible.

6.7.3 Version du langage Java supporté

Sur les 201 différentes instructions *bytecode* d'une JVM « classique », JOP en implémente 43 en tant qu'instruction microcode unique, 92 en tant que suite d'instructions microcode, et 41 en Java. Les instructions restantes sont implémentées par des méthodes natives codées en C.

Ce processeur permet une implémentation suivant la spécification CLDC 1.0.

6.7.4 Evolution

Au mois de Juin 2009, les dernières modifications apportées à JOP dataient de Mars 2009 et des groupes de discussions avec le développeur de JOP, sont actifs (avril 2009). Ce projet n'est donc pas obsolète et on recense aujourd'hui 11 projets académiques et 4 projets industriels qui l'utilisent [15].

6.8 Jazelle

6.8.1 Historique

Jazelle est la technologie mise au point par ARM pour pouvoir exécuter directement du *bytecode* Java au niveau matériel. Elle fut intégrée pour la première fois en 2002 dans le processeur ARM926EJ-S. On retrouve également cette technologie dans le processeur ARM1176JZF (utilisé notamment dans l'iPhone), ainsi que dans tous les processeurs ARM dont la référence contient la lettre 'J'.

L'atout de Jazelle réside dans son intégration à des processeurs génériques, sachant qu'en 2002 les différents modèles ARM équipaient déjà les trois quarts des terminaux mobiles. Jazelle est une solution moins onéreuse et plus simple à intégrer que les processeurs Java dédiés. En effet, cette technologie vient directement s'intégrer au cœur des processeurs existants. Elle ne nécessite donc pas, contrairement aux co-processeurs « indépendants », de modification matérielle et ne remet pas en cause ce que le processeur était capable d'exécuter auparavant. De plus, cette technologie peut bénéficier du « cache » du processeur dans lequel elle est intégrée et permet de s'affranchir de l'utilisation du bus de données utilisé normalement pour l'échange de données entre le processeur et le co-processeur.

15. `http://www.jopwiki.com/Projects`

6.8.2 Implémentation

Jazelle permet d'exécuter directement au niveau matériel environ 90% d'un programme Java classique.

Les spécifications de l'interface matérielle sont intentionnellement très incomplètes. En effet, l'objectif est d'être capable d'implémenter des programmes qui seront supportés par la JVM qu'utilise Jazelle. Seule la JVM a besoin d'avoir (ou est autorisée à avoir) accès aux détails de l'interface matérielle.

Jazelle propose deux technologies :

- ARM Jazelle DBX (*Direct Bytecode eXecution*). Cette première technologie permet l'exécution de *bytecode* directement au niveau matériel ;
- ARM Jazelle RCT (Runtime Compilation Target). Utilisée en complément de Jazelle DBX, cette technologie apporte le support de la compilation *AOT* et *JIT* pour Java, mais également pour .NET, Python et Perl.

Jazelle DBX peut donc être utilisée seule pour des systèmes où l'on dispose de peu de ressources ou alors quand la durée d'une compilation *AOT* et *JIT* serait trop importante. On peut également utiliser les deux technologies simultanément pour, par exemple, bénéficier d'un temps de démarrage plus rapide, ou alors n'utiliser que la technologie Jazelle RCT.

Les compilateurs Java produisent du *bytecode* « générique » qui n'est donc pas optimisé pour une cible spécifique. La technologie Jazelle DBX ne fait de différence entre du code optimisé ou non-optimisé, elle se contente d'exécuter les séquences de *bytecode* qu'on lui fournit. ARM a donc introduit *Jazelle Runtime Optimizer* (JRO) qui réalise, à l'exécution, de la traduction Java *bytecode* vers Java *bytecode*. JRO déplie (*inlining*) les méthodes les plus utilisées, limite les blocages de *pipeline* [16], et transforme le *bytecode* exécuté de façon logicielle en du *bytecode* exécutable par le matériel.

La nouvelle machine virtuelle multi-tâches de Jazelle [1] permet d'exécuter plusieurs applications simultanément. Les limites du tas de chaque application peuvent être ajustées pendant leur exécution. Lorsqu'une application a un besoin supérieur à ce qui est disponible, l'erreur *out-of-memory* n'est pas immédiate. En effet, un composant appelé *Application Management System* (AMS), qui gère les quotas, est informé que l'application a un besoin trop grand. L'AMS peut alors terminer l'application ou l'informer [17] qu'elle doit réduire son utilisation mémoire. C'est seulement si l'application continue d'essayer d'utiliser plus de mémoire que l'erreur *out-of-memory* apparaît.

16. Essentiellement, un *pipeline* utilise l'unité arithmétique et logique (ALU) pour exécuter plusieurs instructions en même temps et en différentes étapes. Les conflits dans le *pipeline* se produisent quand une instruction dans une étape de le *pipeline* dépend du résultat d'une autre instruction qui est devant elle dans le *pipeline* et qui n'est pas encore totalement exécuté. Ces conflits causent un blocage du *pipeline* (« *Pipeline Stall* ») et une perte de temps pendant lequel le processeur attend la résolution du conflit. Les optimisateurs peuvent ordonnancer et réorganiser les instructions pour éviter au mieux les blocages du *pipeline*.

17. Le mécanisme de notification utilisé ne fait pas partie des informations publiquement disponibles.

La documentation d'ARM explique que chaque application s'exécute comme si elle était sur sa propre machine virtuelle et en étant isolée des autres applications. Il est indiqué que ARM utilise le concept sur les « applications multiples isolées » défini dans le *CLDC HotSpotTM Implementation 2.0*, mais aucun détail sur son implémentation n'est disponible.

6.8.3 Version du langage Java supporté

Jazelle est réalisée suivant la spécification CLDC relative à la plate-forme J2ME. Mais, ce processeur peut également être utilisé pour accélérer des applications J2SE comme par exemple lors de l'utilisation du système d'exploitation SavaJe XE.

6.8.4 Evolution

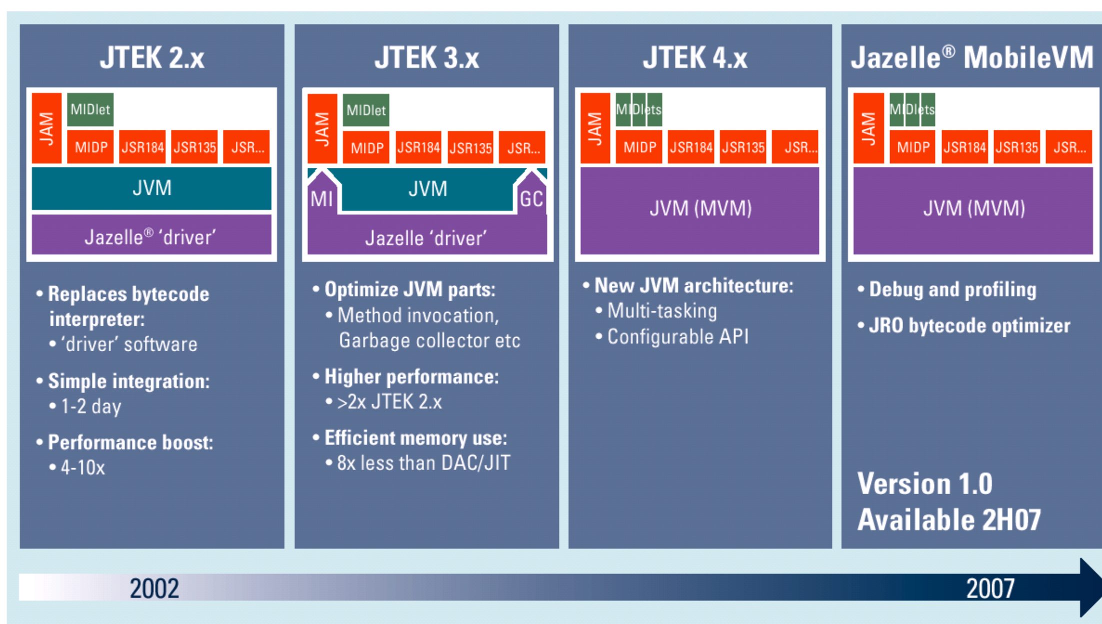


FIGURE 12 – Evolution de Jazelle

La première version de Jazelle n'était qu'un *driver* partiel qui améliorait les performances des processeurs compatibles avec Jazelle DBX (*Direct Bytecode eXecution*).

Ensuite, les optimisations de la JVM, comme celles effectuées au niveau du *garbage collector* ou de l'exécution des appels des méthodes, ont doublé les performances de la version 3 de Jazelle tout en diminuant l'utilisation de la mémoire (8 fois moins qu'un simple compilateur *JIT*).

Par la suite, la nécessité d'une plate-forme Java multi-tâches a entraîné la restructuration de la JVM de Jazelle. En collaboration avec, entre autre, Qualcomm, Sprint, Aplix et les 5 principaux industriels en téléphonie mobile, ARM a définit une API dédiée aux multi-tâches et a implémenté la MVM (*Multi-tasking Virtual Machine*).

6.9 Autres implémentations

Il existe d'autres processeurs Java qui ont existé ou existent toujours, mais, au sujet desquels, nous disposons de très peu d'information.

Une liste (non exhaustive) des processeurs / co-processeurs est :
- Espresso et DeCaf : processeur et co-processeur réalisés par Aurora VLSI en 2001 ;
- LavaCORE [18] : processeur actuellement réalisé par LavaCORE ;
- Lightfoot [19] : réalisé par DCT, ce processeur est intégré au VS2000 de Velocity et couvre 100% du *bytecode* ;
- Moon : réalisé par Vulcan machines en 2000, ce processeur Java pouvait être utilisé seul ou en tant que co-processeur.
- OC104J [20] : processeur actuellement réalisé par Vivaja et possédant 4 cœurs ;
- PCS1000/Ignite [21] : processeur Java réalisé par PTSC ;
- XPRESSOcore : processeur réalisé en 2000 par Zucotto Wireless.

Une liste (non exhaustive) des projets universitaires est :
- BlueJEP : processeur Java développé par l'université de Vienne ;
- FemtoJava : processeur Java développé par l'université de Rio ;
- JaMuth [22] : anciennement « Komodo », c'est un processeur développé par l'université de Augbourg depuis 2001 ;
- jHISC [23] : processeur Java développé par l'université de Hong-Kong ;
- SHAP [24] : processeur développé par l'université de Dresde depuis 2006.

6.10 Tableau de synthèse

18. http://www.lavacore.com/products_ip.htm
19. http://www.velocitysemi.com/
20. http://www.vivaja.com/
21. http://www.ptsc.com/
22. http://www.informatik.uni-augsburg.de/en/chairs/sik/publications/papers/2007_jtres_uhr.html
23. http://www.ee.cityu.edu.hk/~hisc/architecture.html
24. http://shap.inf.tu-dresden.de/

	Fournisseur	Année de lancement	Activité	Type	Standard Java
aJile	Ajile Systems	1999	Ac / I	Processeur	J2ME - CLDC - CDC
AU-J1000(Espresso)	Aurora VLSI	2001	I	Processeur / Co-processeur	*bytecode* couvert à 100%
AU-J1100 ;AU-J1200(DeCaf)				Co-processeur	-
IM1000/IM3000	IMSYS	1999	Ac / I	Processeur	J2ME - CLDC
Jazelle	ARM	2002	Ac / I	Co-processeur	J2ME - CLDC / J2SE
JSTAR/JA108	Nazomi	-	Ob / I	Co-processeur	J2ME - CLDC *bytecode* couvert à 70%
JOP	Martin Schoeberl	2004	Ac / A / I	Processeur	-
LavaCORE	LavaCore	-	Ac / I	Processeur	-
Lightfoot	DCT	-	I (Intégré au VS2000 de Velocity)	Processeur	J2ME, *bytecode* couvert à 100%
Moon	Vulcan machines	2000	Ob / I	Processeur / Co-processeur	J2ME - CLDC
OC104J	Vivaja	-	Ac / I	Processeur (quad core)	-
picoJava	Sun	1997	-	Processeur	Complet
PSC1000/Ignite	PTSC	-	I	Processeur	-

FIGURE 13 – Synthèse de l'étude des processeurs Java - *Légende : A Académique, I Industriel, Ac Actif, Ob Obsolète*

7 MODÈLE D’EXÉCUTION JAVACARD

Cette section se veut une introduction au modèle d’exécution des applets Java qui s’exécutent sur des cartes à puces, communément appelées JavaCard. On s’intéresse ici à la spécification JavaCard version 2, qui représente encore la majorité de la base installée, sachant que la version 3 de la spécification n’est disponible que depuis avril 2008. L’analyse est axée sur les principales différences entre un environnement d’exécution JavaCard et un environnement d’exécution Java plus classique, tel que celui trouvé dans un environnement Java 2 Micro Edition (J2ME), Java 2 Standard Edition (J2SE), ou Java 2 Enterprise Edition (J2EE).

Cette analyse reste valable pour JavaCard version 3 dit « *Classic Edition* » [25], qui est une mise à jour mineure de JavaCard version 2.

L’environnement d’exécution JavaCard n’est pas un environnement Java complet, mais un environnement minimaliste destiné à faire tourner des « applications » (appelées applets) Java dans un environnement d’exécution disposant de ressources matérielles très réduites. C’est ainsi que la première version de la spécification JavaCard était conçue pour fonctionner sur des microprocesseurs disposant de seulement 256 octets de mémoire vive.

7.1 Un sous ensemble du langage Java

Le langage Java utilisé pour la programmation des applets Javacard n’est pas le langage Java standard, mais constitue un sous ensemble strict et cohérent de celui-ci. Ceci veut dire que l’ensemble des mots-clés et fonctionnalités du langage Java pour l’environnement JavaCard existent dans J2SE, mais pas l’inverse. Le *bytecode* Java d’une application JavaCard pourrait donc en théorie être exécuté dans un environnement J2SE, pourvu que les classes de base de l’environnement JavaCard soient portées sur cet environnement.

Les fonctionnalités suivantes du langage Java ne sont pas supportées dans l’environnement JavaCard :

- les types primitifs de grande taille : `long, double, float` ;
- les types énumérés ;
- les chaînes de caractères et le type caractère ;
- les tableaux de dimension plus grande que 1 (seuls les tableaux uni-dimensionnels sont supportés) ;
- le chargement de classes à la volée ;
- le *Security Manager* ;

25. L’autre édition de JavaCard 3, la « *Connected Edition* » est un portage de la JVM de JavaME CLDC, à laquelle est adjointe la bibliothèque de classe de JavaCard.

- le glaneur de cellules et la finalisation d'objets ;
- les *threads* ;
- la sérialisation ;
- le clonage d'objets (les allocations mémoires sur le tas ne sont autorisées que dans la phase d'installation de l'applet).

Les fonctionnalités suivantes du langage Java sont supportées de manière optionnelle :
- le support des entiers sur 32 bits (type de données `int`)

7.2 Cycle de vie d'une applet

Une applet commence sa vie en étant installée sur une JavaCard. L'installation provoque l'exécution de la méthode `install` de l'applet, qui a alors l'occasion d'initialiser ses structures de données internes. C'est le seul moment où l'applet est autorisée à faire de l'allocation dynamique de mémoire. Ensuite, l'applet est sollicitée par l'environnement Javacard pour traiter les requêtes (messages de type APDU) qui lui sont destinés.

La mémoire est finalement libérée, lorsque l'applet est supprimée (désinstallée) de la JavaCard.

7.3 Dynamique de fonctionnement

Une applet Java est uniquement passive. Elle reçoit une requête, calcule le résultat, puis envoie sa réponse. C'est la JCVM qui est responsable de routage des messages entre les différentes applets. Une seule applet peut être en cours d'exécution à un instant t.

7.4 Une machine virtuelle adaptée

Le mode d'exécution sur une JavaCard se rapproche d'un mode d'exécution sur machine virtuelle. En effet, il existe une machine virtuelle appelée JCVM, pour *JavaCard Virtual Machine*, permettant d'exécuter du *bytecode* sur la carte à puce. Il est intéressant de noter que le *bytecode* exécuté par cette machine virtuelle n'est pas du *bytecode* Java standard, mais un *bytecode* spécialement adapté pour réduire l'encombrement mémoire.

Ce *bytecode* Java non standard est obtenu par l'utilisation d'un convertisseur à partir d'un fichier de *bytecode* standard. La conversion résulte en la génération d'un fichier CAP, pour ***C**onverter **AP**plet*. C'est ce format de fichiers qui est interprété par la JCVM.

La vérification de *bytecode* a bien lieu et est effectuée par le convertisseur *bytecode* vers CAP.

Il est à noter qu'il existe quelques curiosités intéressantes au niveau du fonctionnement JavaCard :

- le chargement de classes a lieu lors de l'étape de conversion. Ceci revient à dire que le code Java s'exécutant dans la JCVM n'a pas accès à un chargeur de classes ;
- la persistance. Les objets sont créés en zone de mémoire non volatile. On peut forcer la création de certains objets sensibles en RAM si on le souhaite ;
- le support transactionnel et opérations atomiques. Toutes les opérations d'écritures dans un champ d'un objet sont atomiques. Ceci veut dire qu'en cas de coupure de courant, on sait que l'écriture a eu lieu totalement ou pas du tout. De plus, il est possible de définir des zones spécifiques du code devant constituer une transaction, et donc être exécutés totalement ou pas du tout ;
- la JCVM est démarrée lors de la fabrication de la carte, puis ensuite ne s'arrête jamais. Lorsque la carte n'est pas alimentée électriquement, la JCVM est seulement en pause. Le support des transactions permet à la JCVM de se remettre dans un état stable lorsque son alimentation électrique est rétablie : on appelle cette opération un *card reset* ;
- l'isolation. La JCVM autorise l'exécution de plusieurs applets en son sein. Les applets sont protégées les unes des autres, et ne peuvent pas interférer entre elles.
- pas de *Security Manager*. L'architecture de sécurité de JavaCard ne repose pas sur l'utilisation du *Security Manager*.

De la même manière que dans le monde Java standard, le mécanisme par lequel les applets sont exécutées n'est pas imposé par la spécification. Celles-ci peuvent être interprétées ou compilées à la volée.

7.5 Mécanismes de sécurité

De par sa vocation à faciliter le développement d'applications de sécurité pour les environnements cartes à puce, l'environnement JavaCard inclut un grand nombre de mécanismes de sécurité.

Tout d'abord, l'environnement JavaCard, constitué de convertisseurs d'applets et de la JCVM, inclut les mêmes mécanismes que Java en ce qui concerne :

- la vérification des indices utilisés pour l'accès aux tableaux ;
- l'absence d'arithmétique de pointeurs et le typage fort ;
- le vérificateur de *bytecode*.

La différence la plus notable est que la JCVM n'inclut pas de vérificateur de *bytecode*. La vérification du *bytecode* est effectuée par le convertisseur d'applets.

Elle inclut également les mécanismes suivants qui, bien que n'étant pas à proprement parler des mécanismes de sécurité, facilitent l'écriture d'applications correctes :

– les modificateurs de visibilité [26] ;
– le regroupement de classes dans les *packages*.

JavaCard apporte en sus les mécanismes suivants :

– support transactionnel et opérations atomiques ;
– le *firewall* d'applets, qui empêche une applet JavaCard d'aller empiéter sur les autres applets s'exécutant sur la même carte, sauf si une applet choisit explicitement de rendre un de ses objets accessible pour les autres applets [27] ;
– l'accès aux capacités cryptographiques de la carte à puce (chiffrement/déchiffrement avec algorithmes symétriques ou asymétriques, générateurs de nombres aléatoires, résumés cryptographiques...).

Il est intéressant de noter qu'une fois qu'une applet est transformée au format CAP, le respect de l'intégrité de cette applet avant son chargement sur la JavaCard est primordial, car il n'y a plus de vérification faite une fois l'applet sur la JavaCard : la JCVM ne contient pas de vérificateur de *bytecode*.

La spécification JavaCard ne précise pas quel mécanisme sera utilisé pour vérifier l'intégrité des applets, celui-ci est laissé à la discrétion de l'organisme qui diffuse les cartes.

Une fois l'applet installée, la protection de l'intégrité du *bytecode* transformé est assurée par l'environnement d'exécution final (la JavaCard). L'implication en termes de sécurité est donc moindre que dans l'environnement d'exécution natif étudié en section 5, où la problématique est encore présente sur l'environnement de déploiement (de type PC), et qu'on peut considérer comme moins sûr que la JavaCard.

26. Il est important de noter que même en l'absence de *Security Manager*, il n'est pas possible de contourner les modificateurs de visibilité par introspection, ce mécanisme n'étant pas supporté par JavaCard

27. Ce paragraphe se voulant une introduction à JavaCard, le fonctionnement du *firewall* ne sera pas détaillé ici.

8 PROBLÉMATIQUE DE LA DÉCOMPILATION

La compilation d'un code source Java suivant le modèle standard produit un langage intermédiaire : le *bytecode*. Contrairement aux langages tels que C ou C++, l'approche de Java offre un avantage considérable en termes de portabilité, puisque le *bytecode* généré est exécutable par des JVM fonctionnant sur des plates-formes d'exécution natives différentes. Néanmoins, dans le but de permettre une telle portabilité, le niveau d'abstraction du *bytecode* généré doit rester suffisamment élevé. La conséquence directe de cette abstraction concerne la relative similarité du code généré par rapport au code source. Comme nous le verrons plus en détails dans la section 8.2, cette similarité a un impact direct sur la capacité de protection de Java contre la rétro-ingénierie.

Or, un développeur peut vouloir protéger son programme Java contre cette menace de rétro-ingénierie. Trois besoins sont généralement évoqués pour justifier la mise en place de protections face à une telle menace :

- protéger la propriété intellectuelle et/ou les secrets industriels présents dans les classes Java ;
- protéger l'application contre la recherche de vulnérabilités. L'objectif est d'empêcher un attaquant de trouver et d'exploiter une vulnérabilité présente dans le programme Java ;
- enfouir une clé de chiffrement ou un mot de passe dans le *bytecode*.

Les différents mécanismes de protection de codes natifs (c'est-à-dire issus de la compilation vers le langage machine de la plate-forme d'exécution native) sont nombreux. Les familles de protection techniques les plus connues sont :

- l'offuscation : technique visant à rendre un programme incompréhensible par un humain, tout en restant fonctionnel ;
- l'anti-débogage : technique visant à rendre un programme difficile à analyser de manière dynamique ;
- le *packing* : technique visant à protéger des portions du code binaire avec de la compression et/ou du chiffrement, la décompression et le déchiffrement se faisant lors de l'exécution ;
- l'*anti-dump* : technique visant à empêcher la récupération de l'image mémoire d'un processus pendant son exécution ;
- l'exécution dans un environnement de confiance (par exemple, des parties de *bytecode* sont fournies chiffrées, et sont déchiffrées puis exécutées sur un serveur distant ou sur une *smart card* dédiée).

Certains outils professionnels existent et implémentent une ou plusieurs de ces techniques. Ils peuvent offrir un niveau de robustesse satisfaisant dès lors que le but recherché est de limiter la capacité d'ingénérie inverse aux seuls experts disposant d'un temps de recherche conséquent. Bien qu'il n'existe aucune technique offrant une protection totale d'un programme, la complémentarité des mécanismes peut s'avérer efficace pour certains usages (notamment lorsque le

mécanisme de protection est renouvelé régulèrement). Toutefois, le niveau de protection offert par certaines de ces techniques (en particulier, l'offuscation) n'est pas comparable à celui proposé par les moyens cryptographiques modernes (dès lors que l'attaquant n'a pas accès aux clés de chiffrement et que les mécanismes sont correctement implémentés). De plus, les protections reposent généralement sur des techniques empiriques et le domaine souffre d'un manque de formalisation permettant de coter quantitativement la résistance des mécanismes.

Même si l'efficacité de ce type de mécanisme dépend du type d'application protégée et du but recherché, il peut s'avérer intéressant dans certains cas de les utiliser pour protéger des applications Java. Ce besoin soulève un certains nombre de questions :

- peut-on utiliser de tels mécanismes pour protéger une application Java distribuée sous forme de *bytecode* exécuté par une JVM ?
- qu'en est-il de la résistance relative de ces mécanismes sous Java (au regard de leurs homologues pour le code natif) ?

Il convient tout d'abord de noter que ces techniques ne sont pas toutes applicables aux programmes Java distribués sous forme de *bytecode* exécuté par une JVM. En réalité, seuls l'offuscation, l'exécution dans un environnement de confiance et un dérivé du *packing* — le chiffrement des classes — peuvent être mis en œuvre dans le cadre de programmes Java. Cette dernière technique nécessite, en outre, l'utilisation d'un *class loader* spécifique. Malheureusement, cette approche est clairement remise en question par l'absence de protection du *bytecode* projeté en mémoire. Ceci est d'autant plus vrai depuis l'apparition des techniques d'instrumentations sous Java 1.5 (le *package* `java.lang.instrument` de la bibliothèque standard fournit notamment des facilités permettant d'obtenir le *bytecode* de toutes les classes chargées par la JVM avant leur initialisation). De ce fait, elle ne sera pas présentée plus en détail dans cette étude. De même, l'exécution dans un environnement de confiance sort du cadre de la présente étude, car elle implique un composant externe. Par conséquent, seuls les mécanismes d'offuscation, qui s'appliquent uniquement sur le *bytecode* Java, seront présentés plus en détail en section 8.2.

8.1 Décompilation de *bytecode*

À première vue, la compilation de code source Java vers du *bytecode* n'apporte pas d'optimisations ayant comme effet de bord de compliquer la tâche de rétro-ingénierie. Ce n'est pas le cas, par exemple, de la compilation vers du code natif. Les principaux compilateurs, tels que GCC, introduisent en effet des optimisations ayant pour effet de modifier l'enchaînement des instructions (fusion, suppression ou encore réordonnancement d'instructions). Ces optimisations ont pour effet de compliquer de manière importante l'opération de décompilation.

La décompilation du *bytecode* généré par un compilateur Java est donc un processus relativement simple, qui produit un code proche du code source original. Par conséquent, le code généré est compréhensible par un développeur Java. Différents outils sont disponibles depuis plusieurs années. Ils ne seront pas décrits dans ce document, mais seront simplement cités en

référence. Les plus connus sont Jad[28], DejaVu[29], SourceAgain[30] et Mocha[31]. De nombreux autres produits existent et se basent le plus souvent sur Jad.

De manière générale, les produits de décompilation peuvent être classés en deux catégories (voir [14]). Il y a d’une part ceux qui présupposent que le *bytecode* a été produit par un compilateur spécifique. Ces décompilateurs commencent par identifier le compilateur utilisé et recherchent ensuite les motifs de compilation associés afin de les inverser. La prise en charge de *bytecode* offusqué pose généralement problème du fait de la difficulté de trouver des motifs. Les outils cités dans le paragraphe précédent font partie de cette catégorie. D’autre part, il y a les décompilateurs qui ne reposent pas sur une telle hypothèse et essaient d’abord d’identifier la structure du flot de contrôle avant de rechercher des motifs génériques. Ils sont plus aptes à traiter du *bytecode* ayant subit des traitements d’optimisation ou d’offuscation. En contrepartie, ils fournissent un code source Java qui peut diverger de manière notable avec le code source original, bien que sémantiquement équivalent.

Voici un exemple montrant la capacité de l’outil Jad à décompiler une portion de *bytecode* Java, non offusquée, issue du code source suivant :

```
public class UneClasse
{
  public int entierClasse;
  protected String chaineClasse;
  private String chaineClassePrivate;

  public UneClasse(int entier, String chaine){
    chaineClasse = chaine;
    entierClasse = init(entier);
  }

  private int init(int entier){
    int entierLocalNI; // non initialise
    int entierLocalNINU; // non initialise et non utilise dans la suite
    int entierLocalNU = 100; // non utilise dans la suite
    int entierLocalI = 10;

    entierLocalNI = entierLocalI + entier;
    return entierLocalNI * 2;
  }

  public static void main(String arg[]){
    UneClasse uc = new UneClasse (33, "test");
    String chaineFinale = uc.chaineClasse;
    System.out.println(chaineFinale);
  }
}
```

28. http://www.kpdus.com/jad.html
29. http://www.isg.de/OEW/Java/
30. http://www.ahpah.com/product.html
31. http://www.brouhaha.com/~eric/software/mocha/

L'outil Jad produit le code décompilé suivant :

```
import java.io.PrintStream;

public class UneClasse{
  public UneClasse(int i, String s){
    chaineClasse = s;
    entierClasse = init(i);
  }

  private int init(int i){
    byte byte0 = 100;
    byte byte1 = 10;
    int j = byte1 + i;
    return j * 2;
  }

  public static void main(String args[]){
    UneClasse uneclasse = new UneClasse(33, "test");
    String s = uneclasse.chaineClasse;
    System.out.println(s);
  }

  public int entierClasse;
  protected String chaineClasse;
  private String chaineClassePrivate;
}
```

Cet exemple simple montre à quel point le code généré est semblable au code source original. Les commentaires et les noms d'identifiants locaux ont été respectivement supprimés et modifiés. La présentation du code est restée la même et apparaît donc facilement compréhensible par un développeur.

D'une manière générale, le processus de compilation-décompilation peut être considéré comme quasi-réflexif. Aucune information n'est perdue dans l'étape de décompilation. Seule la compilation vers le *bytecode* engendre des pertes. Voici l'essentiel des modifications identifiées :

- suppression des variables locales non initialisées et non utilisées ;
- perte des commentaires (sauf si le code a été compilé avec l'option de mise au point) ;
- perte du nom d'origine des variables locales et des arguments des méthodes (sauf si le code a été compilé avec l'option de mise au point) ;
- transformation du type de certaines variables locales afin d'optimiser l'utilisation de la mémoire (exemple : une variable de type `int` peut être transformée en `byte` ou `char` en fonction de la taille des données qu'elle reçoit).

Les pertes d'informations de nommage sont compensées par la capacité des décompilateurs cités en référence à générer des noms de variable pertinents, en se basant notamment sur le nom de la classe dont elles sont issues. Au final, le code généré est facilement compréhensible par un développeur et peut, en outre, être recompilé sans aucun problème. Nous verrons dans les

sections suivantes les techniques d'offuscation qui permettent de compliquer la tâche de décompilation et donc de rétro-analyse. Nous évoquerons également les limites de ces techniques.

8.2 Techniques d'offuscation

La littérature (voir [3], [2] et [13]) identifie différentes approches en ce qui concerne les techniques d'offuscation en Java. Ces approches se différencient par leur degré de robustesse vis-à-vis de la menace de rétro-ingénierie et par leur difficulté de mise en œuvre. Il est ainsi possible de regrouper les techniques d'offuscation suivant quatre catégories :
- l'offuscation de la présentation du *bytecode* ;
- l'offuscation des données ;
- l'offuscation du flot de contrôle ;
- l'offuscation à base de transformations préventives.

Pour chaque catégorie, les principales techniques mises en œuvre sont décrites. Les limitations de ces techniques sont également présentées.

Remarque 3

Il faut noter que notre étude ne se base que sur l'offuscation de bytecode*. Il existe quelques outils d'offuscation de code source, mais ils ne seront pas présentés ici, car ils ne sont pas liés à la problématique de protection contre la décompilation.*

8.2.1 Offuscation de la présentation du *bytecode*

Cette première catégorie d'offuscation permet de rendre difficile la compréhension de la signification du rôle des classes, des méthodes et des attributs. Ces techniques d'offuscation sont les plus simples à mettre en œuvre, et permettent de rendre le *bytecode* décompilé non compréhensible par une simple lecture.

8.2.1.1 Renommage

La technique de renommage consiste à modifier les noms des classes, des méthodes et des attributs de manière à ce qu'un humain ne puisse inférer facilement la signification de ces différents éléments. Il s'agit d'une technique simple qui applique la règle contraire à la bonne pratique de développement qui consiste justement à attribuer des noms explicites à ces différents éléments. La majorité des outils d'offuscation renomme ces identifiants en les réduisant à un

seul caractère. Ceci a comme effet de bord de réduire de manière potentiellement considérable la taille du *bytecode* généré.

Exemple :

```
int monEntier;                => int a;
String maChaine;              => String a2;
Double monDouble;             => Double a3;
public void maMethode(){}     => public void a(){}
```

Limitation Pour des raisons évidentes de compatibilité, les méthodes et attributs publics des classes issues d'API publiques doivent être laissés inchangés. Il est donc important de ne pas offusquer ces méthodes et attributs publics. De plus, il est nécessaire de faire attention à ne pas renommer des objets utilisés dans le cadre d'une programmation réflexive, c'est-à-dire reposant sur la découverte automatisée des méthodes et attributs d'une classe (l'introspection). Les principaux outils d'offuscation proposent ainsi une fonctionnalité pour exclure certains objets du processus d'offuscation.

8.2.1.2 Renommage avec surcharge

Le renommage avec surcharge consiste à renommer les méthodes Java avec un même identifiant, en tirant parti du mécanisme de surcharge. Cette approche permet d'obtenir plusieurs méthodes ayant un nom identique ; la JVM se chargeant de les discriminer en fonction de leur signature.

Exemple :

```
public int methode1(){}                => public int    a(){}
public int methode2(int val){}         => public int    a(int a){}
public String methode3(String str){}   => public String a(String a){}
```

Limitation Cette technique effectue également une forme de renommage des éléments des classes Java. Elle est donc soumise aux mêmes limites que celles évoquées précédemment (section 8.2.1.1). En outre, certaines JVM acceptent des surcharges « agressives » : deux attributs de type différent, mais avec un même identifiant. Cependant, la plupart des compilateurs ne supportent pas de telles surcharges (il s'agit en fait d'une déviation entre la spécification Java et la première version de la spécification du *bytecode* [32], la première interdisant ce type de surcharge mais pas la seconde).

32. Ce problème a été identifié dans la seconde version de la spécification de la JVM. Cependant, il fait seulement l'objet d'une note à la fin du document et on peut s'interroger sur la prise en compte effective de ce point par les implémentations des JVM.

8.2.1.3 Suppression des informations de débogage

Cette technique consiste à modifier et/ou supprimer les informations non nécessaires à l'exécution du *bytecode* Java. Plus particulièrement, il peut s'agir de supprimer la structure `LineNumberTable`, optionnelle, utilisée par les débogueurs et par la JVM dans l'affichage de la pile d'appels des méthodes. La suppression de cette structure complique la phase de reconstruction du code source par les décompilateurs.

8.2.2 Offuscation des données

Les constantes d'un programme, et en particulier les chaînes de caractères, persistent après l'étape de compilation. Ces données apparaissent de manière claire dans le *bytecode*. Leur offuscation est donc nécessaire afin de limiter les moyens de rétro-analyse. La principale technique employée consiste à chiffrer les chaînes littérales.

8.2.2.1 Chiffrement des chaînes littérales

Cette technique consiste à protéger en confidentialité les chaînes de caractères présentes dans le *bytecode* Java. En général, ces chaînes sont particulièrement utiles pour comprendre le rôle d'une méthode. Leur détection au sein du *bytecode* permet en effet d'améliorer la compréhension de la relation entre les différentes classes et méthodes. Le déchiffrement de ces chaînes est réalisé pendant l'exécution à l'aide d'une routine dédiée.

Exemple avant offuscation :

```
public static void main(String arg[]){
  MonChiffreurIP cip = new MonChiffreurIP();
  cip.activate("Chiffrement␣IP␣activé");
  /* ... */
  cip.debug("Mode␣débogage␣activé");
}
```

Exemple après offuscation :

```
public static void main(String a[]){
  MonChiffreurIP a = new MonChiffreurIP();
  a.a("@df4~|*lz£a3ù([:e1!a");
  /* ... */
  a.b("£d@afs4~*5sl£a4w3([zea1");
}
```

Limitation Cette technique est dépendante de la robustesse de la routine de déchiffrement. Elle peut donc être considérée comme efficace contre la rétro-analyse statique (analyse statique sans exécution), mais peu pertinente contre la rétro-analyse dynamique (suivi pas à pas du flot de contrôle pendant l'exécution).

8.2.3 Offuscation du flot de contrôle

Les techniques d'offuscation du flot de contrôle ont pour objectif de rendre plus difficile la rétro-analyse statique du *bytecode* et, dans une certaine mesure, la rétro-analyse dynamique (c'est-à-dire le suivi pas à pas du flot de contrôle pendant l'exécution). Les principaux mécanismes mis en œuvre concernent la modification des sauts conditionnels et la mise à plat de la hiérarchie des classes.

8.2.3.1 Modification des sauts conditionnels

L'analyse statique du *bytecode* peut être rendue plus complexe en modifiant la structure du graphe de flot de contrôle. Pour ce faire, les meilleurs outils d'offuscation implémentent des techniques modifiant le branchement des sauts conditionnels (`switch`, `while`, `for`, `if`, `do`) ou en complexifiant les calculs conditionnels. Ces outils peuvent, par exemple, rajouter des branches non exécutées et des tests conditionnels supplémentaires.

Limitation Une telle méthode d'offuscation peut avoir un impact potentiel sur les performances de l'application. De ce fait, il est important de ne pas l'appliquer aux algorithmes critiques en termes de performances sans évaluation *a posteriori*. Cet effet de bord peut avoir une incidence sur la capacité d'une application à s'exécuter en temps contraint. En particulier, il peut s'agir de préserver les propriétés de sécurité d'un programme, notamment la capacité à résister aux attaques temporelles.

De plus, bien que la modification des sauts conditionnels puisse être considérée comme efficace contre la rétro-analyse statique, cette technique se montre moins pertinente contre la rétro-analyse dynamique. Le suivi pas à pas est simplement rendu plus long en temps.

8.2.3.2 Mise à plat de la hiérarchie des classes

La mise à plat de la hiérarchie des classes a pour objectif de modifier la structure relationnelle entre les classes et les *packages*. Elle met en œuvre des techniques de fusion et de découpage de classes afin de produire une structure « à plat ». Cette approche permet de complexifier

la compréhension du rôle des classes et des relations hiérarchiques issues des mécanismes d'héritage et de polymorphisme.

Limitation Contrairement à la modification des sauts conditionnels, cette technique a l'avantage d'avoir peu d'impact sur les performances. Cependant, elle n'offre pas l'assurance du maintien des propriétés de sécurité du programme, notamment en termes de confidentialité des données. En effet, un attribut `private` peut se retrouver accessible à une classe parente après une mise à plat de la hiérarchie. En revanche, sa mise en œuvre dans un outil d'offuscation est particulièrement complexe. Pour cette raison, les outils du marché ne proposent pas encore une telle fonctionnalité. Cette dernière est néanmoins présente dans des prototypes issus de laboratoires de recherche [13].

8.2.4 Offuscations/transformations préventives

Certains outils mettent en œuvre des techniques d'offuscation visant à tromper un décompilateur spécifique. Par exemple, le fonctionnement de l'outil d'offuscation HoseMocha a pour unique objectif de rendre défaillant l'opération de décompilation effectuée par Mocha. La technique utilisée dans ce cas précis consiste à rajouter des instructions non exécutées après les `return` des méthodes. Ceci n'a aucun impact lors de l'exécution de l'application, mais sera fatale pour Mocha qui tentera de les prendre en compte.

Limitation Peu d'outils mettent en œuvre ce type d'offuscation, puisqu'il est trop dépendant du comportant des décompilateurs ou d'une version spécifique d'un décompilateur. Il s'agit d'une technique essentiellement basée sur l'exploitation d'astuces techniques.

8.3 Analyse des produits d'offuscation

Cette étude présente une analyse des principaux produits d'offuscation du marché, encore maintenus ou en développement à la date de rédaction du présent document :

- Allatori [33] ;
- DashO [34] ;
- Proguard [35] ;
- RetroGuard [36] ;

33. `http://www.allatori.com/features.html`
34. `http://www.preemptive.com/dasho-java-obfuscator.html`
35. `http://proguard.sourceforge.net`
36. `http://www.retrologic.com/retroguard-main.html`

– SmokeScreen [37] ;
– Zelix Klassmaster [38].

8.3.1 Résultats de l'analyse

Le tableau suivant présente, pour chacun des outils identifiés, les types d'offuscation mis en œuvre et, à titre d'indication, les performances mesurées.

Produit		Allatori	DashO	Proguard	Retro Guard	Smoke Screen	Zelix Klass-master
Offuscation de présentation	Renommage	x	x	x	x	x	x
	Renommage avec surcharge	x	x	x	x		x
	Suppression des informations de débogage	x	x	x	x	x	x
Offuscation des données	Chiffrement des chaînes littérales	x	x			x	x
Offuscation du flot de contrôle	Modification des sauts condition-nels	x	x			x	x
	Mise à plat de la hiérarchie des classes						
Transformations préventives				x	x		
Durée d'exécution		+2%	+1%	0%	0%	+1%	+1%
Taille du *bytecode*		+3%	-35%	-32%	-27%	-5%	-24%
Complexité du *bytecode*		+84%	+108%	0%	0%	+115%	+230%

Méthodologie d'analyse Les performances sont présentées suivant trois aspects : la durée d'exécution, la taille [39] et la complexité du *bytecode* généré. La complexité est donnée par le nombre cyclomatique de McCabe [40] qui représente le nombre de chemins indépendants dans le graphe de flot de contrôle du programme. Un accroissement de 10% de cette complexité (par exemple, 10 chemins en plus) induit un temps proportionnellement plus important de rétro-analyse statique. Les tests ayant conduit aux résultats du tableau ont été menés sur du code Java

37. `http://www.leesw.com/smokescreen/index.html`
38. `http://www.zelix.com/klassmaster/features.html`
39. On considère ici que le *bytecode* est vidé de ces informations de débogage avant application des produits.
40. `http://en.wikipedia.org/wiki/Cyclomatic_complexity`

hétérogène : algorithmes, interfaces graphiques, jar multiples. Les données du tableau correspondent au ratio :

$$\frac{\text{valeur après offuscation} - \text{valeur avant offuscation}}{\text{valeur avant offuscation}} \times 100$$

Il pourrait être envisagé de réaliser une offuscation multiple (plusieurs passages d'un même outil ou d'outils différents) dans l'objectif d'augmenter la robustesse du *bytecode* face à la rétro-ingénierie. Cet aspect ne semble cependant pas pertinent, notamment lorsqu'il s'agit d'appliquer une offuscation multiple des données. En effet, la robustesse est identique quel que soit le nombre de renommages.

En ce qui concerne les modifications sur le flot de contrôle, la complexité sera un peu plus importante lors de la seconde passe. Des premiers tests montrent que l'apport limité de résistance dû à une seconde passe n'est pas pertinent face à la menace de rétro-ingénierie. L'essentiel de l'apport en robustesse est fourni lors de la première passe. En d'autres termes, un attaquant capable de « casser » une première passe d'offuscation, sera suffisamment compétent pour attaquer du *bytecode* offusqué à deux reprises.

La combinaison de plusieurs outils apparaît plus pertinente en termes d'augmentation de la robustesse ; mais là encore, la relative similarité des fonctions d'offuscation des outils du marché fait que l'apport d'une seconde passe par un autre outil n'est pas déterminante.

8.3.2 Démarche d'utilisation et limitations

La démarche pour appliquer un programme d'offuscation sur du *bytecode* Java est présentée ci-dessous. Cette démarche est relativement similaire quel que soit le programme utilisé.

1. Compilation du code source en *bytecode*.
2. Configuration des règles d'offuscation (au format XML ou dans une interface graphique).
3. Choix des archives `.jar`, des *packages* ou des classes concernées par l'offuscation.
4. Localisation du point d'entrée de l'application Java, afin de le prendre en compte correctement lors du processus d'offuscation.
5. Choix des éléments (méthodes, attributs) exclus du processus d'offuscation. Cette étape permet de choisir explicitement de ne pas appliquer de transformations sur des éléments sensibles ou sur lesquels l'offuscation n'est pas utile.
6. Choix des types d'offuscation à appliquer (offuscation des données, du flot de contrôle, etc.). Pour les modifications de nommage, il est possible de préciser l'ensemble des noms acceptés (à partir d'un dictionnaire par exemple).
7. Choix du niveau de robustesse des méthodes d'offuscation. Ceci permet de mettre en place, par exemple, des surcharges « agressives ».

8. Lancement du processus d'offuscation. Les outils produisent en sortie une version offusquée du *bytecode* Java, de même qu'un fichier contenant la trace de l'ensemble des transformations.
9. Applications des tests logiciels (tests unitaires, tests d'intégration, etc.) et validation pour vérifier la conformité du comportement du programme.

Les outils du marché ont cependant quelques limitations. D'une part, l'ensemble des techniques théoriques d'offuscation n'est pas disponible avec ces outils, notamment les techniques trop complexes ou ayant un impact trop important sur les performances. De fait, le niveau d'offuscation offert par ces outils s'avère moindre que le niveau de l'état de l'art. D'autre part, le niveau de configuration des outils est relativement inégal. En effet, certains ne permettent pas de choisir explicitement les éléments à exclure du processus d'offuscation. Il convient donc de choisir l'outil en fonction de ses besoins en termes de finesse de configuration. Enfin, ces différents outils ne supportent pas, pour la plupart, les fichiers qui ne sont pas au format Java. Par exemple, les fichiers XML ou les fichiers *properties*, utilisés lors du chargement ou déploiement d'applications Java, et contenant des références vers les éléments de l'application, peuvent ne pas être pris en compte lors du processus d'offuscation. Il convient, là aussi, de choisir un outil capable de prendre en compte de telles situations ou, dans le cas contraire, de propager les renommages d'éléments Java vers les fichiers XML ou fichiers *properties*.

8.4 Conclusion sur la décompilation

La décompilation du *bytecode* Java permet de retrouver avec une précision considérable le code source original. Par conséquent, le fait de distribuer le *bytecode*, sous quelque forme que ce soit (`.class`, `.jar`, etc.), revient à distribuer son code source. Il est donc primordial d'avoir conscience de cette caractéristique, qui n'existe pas pour les programmes en code natif, lorsque l'on souhaite distribuer du code Java.

Il peut alors être envisagé de protéger le *bytecode* contre la menace de rétro-ingénierie. Un développeur souhaitant mettre en œuvre des mécanismes d'offuscation doit cependant en maîtriser les effets et en connaître les limites. D'une part, l'offuscation doit être réalisée par des programmes dont le comportement et les techniques utilisées ont été évaluées. Le maintien de certaines propriétés de sécurité, comme la résistance aux attaques par canaux auxiliaires, ne peut pas être considéré comme assuré par les outils d'offuscation. En outre, la mise en œuvre « à la main » de ces techniques par le développeur est à proscrire. D'autre part, il est nécessaire d'avoir à l'esprit que l'offuscation est une solution qui ralentit la rétro-analyse, mais ne la contre pas. En effet, la décompilation est un processus qui aboutit à différents résultats suivant la robustesse des protections. Si le *bytecode* est faiblement offusqué, le code source généré sera compilable et facilement compréhensible. En revanche, si le *bytecode* est offusqué de manière complexe, certains outils de décompilation vont produire un code source non compilable, mais compréhensible, quant d'autres généreront un code compilable mais plus complexe à appréhender.

Si la menace de rétro-ingénierie est un critère déterminant, il est alors pertinent de compléter ou de remplacer les techniques d’offuscation par d’autres mécanismes de protection. Citons notamment la possibilité d’exécuter du code sensible dans un environnement de confiance (comme par exemple, serveur distant ou carte à puce locale). En ce qui concerne les programmes en code natif, ils disposent d’une large boîte à outils en termes de protections : des mécanismes d’offuscation, mais également des techniques d’*anti-dump*, d’anti-traçage ou encore d’anti-débogage. Ces dernières techniques ne sont cependant pas applicables pour la protection d’un programme Java, notamment à cause du manque de contrôle sur son exécution. Une application Java ne peut donc pas atteindre le niveau de protection d’une application en code natif.

RÉFÉRENCES

[1] Chris Porthouse(ARM) and Dave Butcher(ARM). *Multitasking Java on ARM platforms*, 2006.

[2] Christian Collberg and Clark Thomborson. Watermarking, Tamper-Proofing, and Obfuscation - A Tools for Software Protection. 2002.

[3] Christian Collberg and Clark Thomborson and Douglas Low. A Taxonomy of Obfuscating Transformations. Technical report, The University of Auckland, New Zealand, 1997.

[4] Consortium JAVASEC. Comparatif des compilateurs. Technical Report Livrable 2.1 dans le CCTP JAVASEC, Silicom Région Ouest - Amossys - INRIA Rennes Bretagne Atlantique - SGDN, 2009.

[5] Consortium JAVASEC. Comparatif des JVM. Technical Report Livrable 2.2 dans le CCTP JAVASEC, Silicom Région Ouest - Amossys - INRIA Rennes Bretagne Atlantique - SGDN, 2009.

[6] Consortium JAVASEC. Rapport sur le langage Java. Technical Report Livrable 1.1 dans le CCTP JAVASEC, Silicom Région Ouest - Amossys - INRIA Rennes Bretagne Atlantique - SGDN, 2009.

[7] Imsys Technologies AB. *IM1101C Technical Reference Manual*, 2008.

[8] Imsys Technologies AB. *IM3910 Microcontroller - Datasheet*, 2008.

[9] J. Hummel and A. Azevedo and D. Kolson and A. Nicolau. Annotating the Java bytecodes in support of optimization. *Concurrency : Practice and Experience*, 9(11), 1997.

[10] James Gosling and Bill Joy and Guy Steele and Gilad Bracha. *The Java Language Specification, Third Edition*. Addison-Wesley Longman, Amsterdam, 3 edition, June 2005.

[11] Martin Schoeberl. *JOP : A Java Optimized Processor for Embedded Real-Time Systems*. PhD thesis, Vienna University of Technology, 2005.

[12] Martin Schoeberl and Rasmus Pedersen. WCET Analysis for a Java Processor. 2006.

[13] Micheal Batchelder and Laurie Hendren. Obfuscating Java : the most pain for the least gain . 2006.

[14] Nomair A. Naeem and Laurie Hendren. Programmer-friendly Decompiled Java. Technical report, The University of McGill, Canada, 2006.

[15] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[16] Tom R. Halfhill. IMSYS Hedges Bets On Java. *Microprocessor Report*, 2000.