Nouvelles approches en sécurité de Java

Frédéric Painchaud
RDDC – Valcartier

R & D pour la défense Canada – Valcartier

Note technique
RDDC – Valcartier TN 2002 - 154
2002-11-26

Auteur

Original signé par Frédéric Painchaud

Frédéric Painchaud

Approuvé par

Original signé par Yves Van Chestein

Yves Van Chestein

Chef de Section, Gestion de l'Information et de la Connaissance

Ce travail a été effectué à RDDC – Valcartier en collaboration avec l'Université Laval entre mai et août 2002.

Abstract

In this document, three recently published papers on different aspects of Java Security are summarized. These papers appeared in international conference proceedings and computer science journals. The first paper presents an approach aiming at replacing the dataflow analysis carried out by the Bytecode Verifier in the Java Virtual Machine. This approach is based on model checking. The second paper presents an approach aiming at instrumenting Java bytecode to ensure better security of applets and Jini services. This new approach makes it possible to detect and stop certain types of denial of service, to ensure integrity of critical data and confidentiality of the system and even to thwart certain types of spoofing. Finally, the third and last paper presents a security infrastructure built around an event/response mechanism. This infrastructure has an expressive and rather intuitive security policy specification language.

Since each paper presents a new approach to solve one of the problems of this architecture, it seems to be essential to summarize them in this document in order to underline their application(s). Finally, some of these new approaches' limitations, surmountable or not, and some personal comments are given.

Résumé

Dans ce document, trois articles portant sur différents aspects de la sécurité du langage de programmation Java sont résumés. Ces articles ont été récemment publiés dans des actes de conférences et des journaux internationaux. Le premier article présente une approche visant à remplacer l'analyse de flot de données effectuée par le vérificateur de code objet de la machine virtuelle Java. Cette approche est basée sur le « model checking ». Le deuxième article présente une approche visant à instrumenter le code objet Java afin d'assurer une meilleure sécurité des applets et des services Jini. Cette nouvelle approche permet de détecter et d'arrêter certains types de dénis de service, d'assurer l'intégrité des données critiques et la confidentialité du système et même de prévenir certains types de personnification. Finalement, le troisième et dernier article présente une infrastructure de sécurité utilisant un mécanisme « événement/réponse ». Cette infrastructure possède un langage de spécification des politiques de sécurité expressif et plutôt intuitif.

Puisque chaque article présente une nouvelle approche pour résoudre un des problèmes de l'architecture de sécurité de Java, il semble essentiel de les résumer afin de souligner leur(s) application(s). Pour terminer, quelques-unes des limites, surmontables ou non, de ces nouvelles approches sont données et quelques commentaires personnels sont transmis.

Intentionnellement en blanc.

Executive Summary

This document's primary objective is to present the results of an analysis of certain new approaches in Java Security. This analysis aims at determining if these new approaches are of interest and why they are interesting. Another goal of the analysis is to extract these new approaches' possible applications. Therefore, in this document, three recently published papers on different aspects of Java Security are summarized, after an overview of the actual Java Security Architecture in order to set the analysis' context. These papers appeared in international conference proceedings and computer science journals.

The first paper is the most interesting. It presents an approach aiming at replacing the dataflow analysis carried out by the Bytecode Verifier in the Java Virtual Machine. This approach is based on model checking. The second paper presents an approach aiming at instrumenting Java bytecode to ensure better security of applets and Jini services. This new approach makes it possible to detect and stop certain types of denial of service, to ensure integrity of critical data and confidentiality of the system and even to thwart certain types of spoofing. Finally, the third and last paper presents a security infrastructure built around an event/response mechanism. This infrastructure has an expressive and rather intuitive security policy specification language.

Since each paper presents a new approach to solve one of the problems of the Java Security Architecture, it seems to be essential to summarize them in this document in order to underline their application(s). Furthermore, some of these new approaches' limitations, surmountable or not, and some personal comments are given.

The three papers presented in this document show that Java Security is particularly well designed because the new approaches that have been presented were not all of interest. In fact, the Java Security Architecture only needs to be properly used. Its level of confidence is generally high enough in the context of common information systems. However, for critical and/or exposed systems, the Java Security Architecture must be extended to offer a better level of assurance. Still, these extensions must not sacrifice robustness. Therefore, there is room for research on new approaches in Java Security.

The main stakes in this research sector for the next few years are the fusion of static and dynamic analyses in order to benefit from the advantages of both approaches, the formalization of security systems to achieve better robustness and assurance, the evaluation of residual risk in order to better understand the flaws of security systems and the definition of formal, intuitive and homogeneous security policies to precisely identify acceptable and unacceptable program behaviors.

Painchaud, F. 2002. Nouvelles approches en sécurité de Java. RDDC – Valcartier TN 2002 - 154. RDDC – Valcartier.

Sommaire

L'objectif premier de ce document est de présenter les résultats d'une analyse de quelques nouvelles approches en sécurité du langage de programmation Java. Cette analyse vise à déterminer si ces nouvelles approches sont intéressantes et pourquoi elles le sont. Elle a également pour but de dégager les applications possibles de ces nouvelles approches. Dans ce document, trois articles portant sur différents aspects de la sécurité du langage de programmation Java sont donc résumés, après avoir effectué un survol de l'architecture actuelle de la sécurité de Java, pour situer le contexte d'analyse. Ces articles ont été récemment publiés dans des actes de conférences et des journaux internationaux.

Le premier article est le plus intéressant. Il présente une approche visant à remplacer l'analyse de flot de données effectuée par le vérificateur de code objet de la machine virtuelle Java. Cette approche est basée sur le « model checking ». Le deuxième article présente une approche visant à instrumenter le code objet Java afin d'assurer une meilleure sécurité des applets et des services Jini. Cette nouvelle approche permet de détecter et d'arrêter certains types de dénis de service, d'assurer l'intégrité des données critiques et la confidentialité du système et même de prévenir certains types de personnification. Finalement, le troisième et dernier article présente une infrastructure de sécurité utilisant un mécanisme « événement/réponse ». Cette infrastructure possède un langage de spécification des politiques de sécurité expressif et plutôt intuitif.

Puisque chaque article présente une nouvelle approche pour résoudre un des problèmes de l'architecture de sécurité de Java, il semble essentiel de les résumer afin de souligner leur(s) application(s). De plus, quelques-unes des limites, surmontables ou non, de ces nouvelles approches sont données et quelques commentaires personnels sont transmis.

Les trois articles présentés dans ce document montrent que la sécurité de la plate-forme Java est particulièrement bien conçue, car les nouvelles approches exposées n'ont pas été très concluantes. En fait, l'architecture de la sécurité de Java ne demande qu'à être utilisée adéquatement. Le niveau de confiance qu'elle apporte est généralement assez élevé pour assurer la sécurité d'un environnement courant. Cependant, pour les systèmes critiques et/ou particulièrement exposés, l'architecture de la sécurité de Java doit être étendue pour offrir une plus grande assurance. Toutefois, ces extensions ne doivent pas sacrifier la robustesse. Il reste donc encore de la place pour la recherche dans le domaine de la sécurité de Java.

Pour conclure, les principaux enjeux à surveiller durant les prochaines années dans ce domaine de recherche sont la fusion des analyses statiques et dynamiques afin de jouir des avantages des deux approches, la formalisation des systèmes de sécurité pour atteindre une plus grande robustesse et une meilleure confiance, l'évaluation du risque résiduel pour mieux connaître les faiblesses des systèmes de sécurité et les politiques de sécurité formelles, intuitives et uniformes pour définir avec précision les comportements acceptables et inacceptables des logiciels.

Painchaud, F. 2002. Nouvelles approches en sécurité de Java. RDDC – Valcartier TN 2002 - 154. RDDC – Valcartier.

Table des matières

Intentionnellement en blanc.

1 Introduction[1]

Ce document constitue une synthèse de trois articles [Basin et al.02], [Chander et al.01] et [Hashii et al.00b] portant sur différents aspects de la sécurité du langage de programmation Java (aussi appelée *sécurité de Java*) et récemment publiés dans des actes de conférences ou des journaux internationaux. Le fil conducteur utilisé pour créer ce document synthèse est formé des liens que ces articles possèdent avec l'architecture actuelle de la sécurité de Java [Lindholm & Yellin99]. Puisque ces articles présentent une nouvelle approche à un problème donné de cette architecture, il paraît essentiel de les résumer dans ce document afin de cibler leur(s) application(s). De là, quelques-unes des limites, franchissables ou non, de ces approches ainsi que certains commentaires personnels seront fournis.

Cependant, avant d'étudier le premier article, il est de mise de situer, dans la section 2, le contexte d'analyse en résumant quelque peu l'architecture actuelle de la sécurité de Java. Les sections 3, 4 et 5 présentent les articles étudiés. Finalement, la section 6 conclut ce document et donne quelques remarques d'intérêt général.

Ce travail a été effectué à RDDC – Valcartier en collaboration avec l'Université Laval entre mai et août 2002.

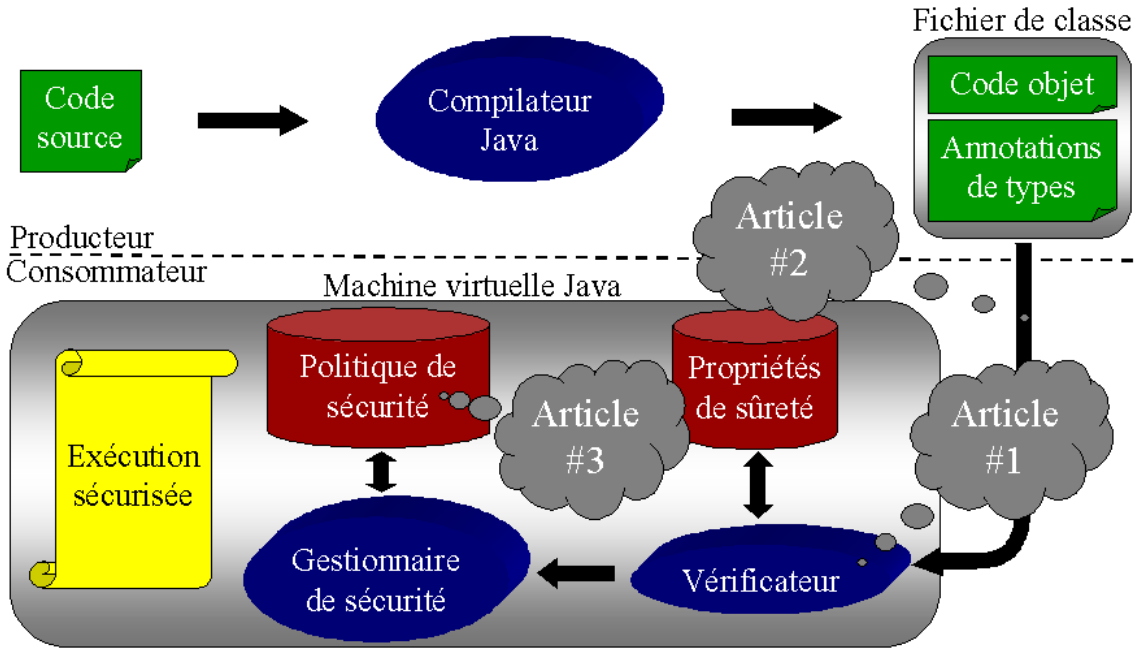
2 Architecture actuelle de la sécurité du langage de programmation Java

La sécurité de Java est assurée à deux niveaux principaux : à la compilation et à l'interprétation. Bien que la sécurité assurée au niveau du compilateur soit très intéressante, sa compréhension n'est pas nécessaire à la lecture de ce document. Elle ne sera donc pas traitée dans ce dernier.

Par contre, la sécurité assurée au niveau de la machine virtuelle Java est directement touchée par les trois articles synthétisés. Elle est donc résumée dans les quelques pages qui suivent de façon à situer le lecteur. Ce dernier est invité à référer à la figure suivante afin d'obtenir une idée générale de l'architecture actuelle de la sécurité de Java et de la place que les trois articles étudiés occupent dans celle-ci :

---

[1] La présentation donnée sur le contenu de ce document est disponible sur le CD-ROM de ce document et peut être consultée en cliquant ici.



2.1 Langage de programmation Java

Le langage Java est largement utilisé de nos jours. C'est un langage orienté-objet reconnu pour sa simplicité, sa portabilité et sa robustesse. À l'origine, il a surtout été conçu dans l'optique de créer des applications ou des applets pouvant être facilement déployées sur Internet. Il supporte maintenant plusieurs domaines, passant par la téléphonie et les systèmes embarqués et répartis.

Ce langage est d'abord compilé afin de générer des fichiers de classe (appelés « class files » en anglais) ayant l'extension *.class*. Ces fichiers sont totalement indépendants de l'architecture de l'ordinateur qui les accueille. Ils contiennent le code objet, appelé « bytecode » en anglais, qui implante chaque méthode déclarée dans une classe.

Ce code objet est ensuite exécuté, ou interprété, par la machine virtuelle Java (MVJ). Puisque cette machine virtuelle peut être développée pour plusieurs plates-formes différentes, comme par exemple Windows et UNIX, les programmes écrits en Java et compilés en fichiers de classe peuvent être exécutés, et ce sans même devoir les recompiler, sur n'importe quel ordinateur utilisant une de ces plates-formes. C'est le concept « Write once, run anywhere », élégamment matérialisé par Java.

Comme Java visait essentiellement l'implantation d'un système à code mobile lors de son apparition officielle en 1995, la garantie de sa sécurité était impérative. Il a donc été élaboré avec cette idée en tête : personne ne veut voir l'Internet gagner en versatilité au risque de perdre des informations cruciales ou de les voir s'étendre à travers celui-ci comme une traînée de poudre.

Java est devenu le standard en matière de contenus exécutables sur Internet et en fait donc une cible de choix pour les utilisateurs malicieux. Ainsi, sa sécurité demande d'être vérifiée et même étendue afin de prévoir les problèmes potentiels et ainsi pouvoir les prévenir. La première étape consiste à comprendre les différents mécanismes en fonction. Les principaux aspects de la sécurité de Java, présents au niveau de la MVJ, c'est-à-dire du code objet, sont donc présentés d'une façon concise.

2.2 Sécurité au niveau de la machine virtuelle Java

Malgré toutes les vérifications effectuées par le compilateur Java et autres mécanismes présents au niveau du langage, il n'en demeure pas moins qu'il y a toujours des possibilités d'attaques de la MVJ en utilisant un compilateur comportant des erreurs ou des malices. Il ne faut pas oublier que la machine virtuelle n'a pas de compilateur intégré et n'utilise donc pas les fichiers contenant le code source Java afin de les compiler avant de les exécuter. Elle s'attend plutôt à recevoir directement des fichiers de classe. Ainsi, à prime abord, elle n'a pas la capacité de savoir si ces fichiers de classe ont été produits par un compilateur correct ou carrément générés à la main par un utilisateur malicieux espérant l'exploiter.

Un autre problème qu'entraîne la vérification effectuée seulement au moment de la compilation est une confusion potentielle au niveau des versions des classes compilées. En effet, pour dérouter les vérifications faites sur les modificateurs, par exemple, un programmeur pourrait compiler une première version d'une classe, nommée *ClasseContenantAttributsPrives*, comportant des attributs déclarés privés et ensuite modifier l'accès à ces attributs en les déclarant publics de façon à pouvoir compiler avec succès une autre classe, nommée *ClasseUtilisantLesAttributs*, par exemple, qui utilise directement ces derniers. Si la machine virtuelle n'effectuait pas certaines vérifications, elle permettrait ainsi à la classe *ClasseUtilisantLesAttributs* de faire directement référence aux attributs, pourtant déclarés privés, de la classe *ClasseContenantAttributsPrives*.

Tous les fichiers de classe chargés par la machine virtuelle Java sont donc analysés statiquement avant d'être utilisés d'une quelconque façon. C'est le vérificateur de code objet (de l'anglais « Bytecode Verifier ») qui est responsable de cette tâche.

Ce dernier s'assure que la structure des fichiers que la MVJ reçoit en entrée respecte bien la structure définie pour les fichiers de classe. Il effectue aussi une analyse très poussée, appelée analyse de flot de données, sur chaque méthode que chaque fichier de classe contient. Ces vérifications permettent de garantir certaines propriétés de sûreté essentielles à l'élaboration de politiques de sécurité se plaçant à un niveau plus élevé.

Il est bon de noter certains avantages importants inhérents à l'utilisation d'un tel vérificateur :

- *Il améliore les performances de la machine virtuelle*. En effet, s'il était absent, beaucoup de tests devraient être faits au moment de l'exécution des fichiers de classe, c'est-à-dire avant l'exécution proprement dite de chaque instruction que

chacune des méthodes contient. Ainsi, compte tenu des propriétés assurées par le vérificateur, la MVJ peut exécuter les instructions en considérant que les tests nécessaires ont déjà été faits. Ces tests sont donc faits statiquement, donc une seule fois, plutôt que dynamiquement, c'est-à-dire pour chaque exécution, ce qui est beaucoup plus efficace.

- *Il est indépendant du compilateur*. Il est entendu ici qu'il a la capacité de vérifier et de certifier du code qui n'a pas été généré par un compilateur Java. En fait, si le code vérifié est correct, conformément à certaines propriétés détaillées aux sous-sections 2.2.1 et 2.2.2, il passera les examens avec succès et pourra être exécuté par la MVJ.

- *Il est indépendant du langage Java*. Puisqu'il vérifie le code objet et non pas directement le code source Java, il n'empêche pas d'autres langages d'être compilés en fichiers de classe afin d'être exécutés par la machine virtuelle.

- *Il est extrêmement conservateur*. Par conservateur, il est entendu qu'il refusera de certifier certains fichiers de classe qu'un vérificateur plus sophistiqué pourrait certifier avec succès. Les concepteurs de Java ont en effet tenté de faire en sorte que le vérificateur demeure le plus simple possible (étant déjà passablement compliqué). Au prix d'une plus grande complexité, ils auraient pu, par exemple, inclure certains tests permettant d'effectuer une vérification plus fine au niveau de certaines structures du code objet mais ont simplement décidé de refuser les fichiers de classe contenant ces structures.

Les détails des différents tests effectués par ce vérificateur, c'est-à-dire principalement l'examen de la structure des fichiers de classe et la réalisation d'une analyse de flot de données sur le code de chaque méthode que contient chaque fichier de classe, sont maintenant présentés.

2.2.1 Structure des fichiers de classe

Les fichiers de classe ont une structure, très bien définie dans [Lindholm & Yellin99], qui permet de les distinguer facilement des autres fichiers. Il ne faudrait en aucun cas que la machine virtuelle soit sensible à des attaques qui consisteraient simplement, par exemple, à renommer un fichier quelconque pour lui donner l'extension *.class*.

Voici un aperçu général de la structure d'un fichier de classe, les différents éléments étant mentionnés en suivant l'ordre dans lequel ils apparaissent réellement dans le fichier :

- le nombre 0xCAFEBABE ;

- le numéro de version du format utilisé ;

- le regroupement de constantes (ou « constant pool » en anglais) ;

- des informations concernant la classe que ce fichier représente ;
- des informations concernant chaque attribut et méthode de cette classe ;
- des informations, sous forme d'attributs, concernant cette classe.

Le nombre 0xCAFEBABE n'est qu'une constante qui doit être présente au début d'un fichier de classe et qui permet de savoir en un clin d'oeil s'il s'agit d'un autre type de fichier.

La structure du fichier de classe étant importante aux yeux du vérificateur de code objet, il doit savoir quelle est la version de la structure du fichier courant afin de déterminer s'il est en mesure de le vérifier et quelle est la batterie de tests à effectuer pour ce faire. C'est donc pourquoi le numéro de version du format utilisé est indiqué.

Comme son nom l'indique, le regroupement de constantes contient toutes les constantes utilisées dans le fichier de classe : les nombres et les chaînes de caractères constants, le nom des attributs et des méthodes, le nom des différentes classes utilisées dans la classe, etc. Ce regroupement prend la forme d'une table dont les éléments peuvent être lus à l'aide d'un indice. Ainsi, aucune autre partie du fichier de classe ne fait référence explicitement à une constante. Toutes ces références sont plutôt faites en spécifiant un indice dans le regroupement de constantes. Ce dernier existe surtout pour rendre les fichiers de classe plus compacts. En effet, plutôt que d'être répétées un peu partout dans le fichier, les constantes ne sont ainsi mentionnées qu'une seule fois, dans le regroupement de constantes.

Chaque élément du regroupement de constantes possède un champ qui révèle son type : nombre, chaîne de caractères, nom d'attribut, nom de méthode, nom de classe, etc. Connaissant son type, il est possible de vérifier si l'élément a la bonne longueur et si les données qu'il contient sont appropriées. Si le vérificateur rencontre un type inconnu, il doit rejeter le fichier de classe correspondant.

Juste après ce regroupement de constantes, on retrouve des informations sur le nom de la classe que le fichier de classe représente, ses différents modificateurs et le nom de sa superclasse[2]. Ces informations sont essentielles, entre autres pour vérifier qu'une classe donnée a bel et bien accès à une autre classe. Pour ce faire, il s'agit simplement d'inspecter ces informations au niveau de cette autre classe.

C'est directement à la suite des précédentes informations qu'on trouve le nom et le type de chaque attribut et méthode de la classe. Le type d'un attribut ou d'une méthode est désigné par une entrée dans le regroupement de constantes qui représente une chaîne de caractères dénommée *signature*. Pour un attribut, cette signature ne définit en fait rien de plus que son type, de façon standardisée. Dans le cas d'une méthode par contre, elle

---

[2] La superclasse d'une classe donnée est son ancêtre immédiat selon la relation d'héritage.

définit le type de chacun de ses paramètres ainsi que le type de sa valeur de retour. On retrouve aussi à cet endroit, dans un fichier de classe, le code objet associé à chaque méthode de la classe. C'est ce code qui sera analysé plus tard lors de l'analyse de flot de données. Toutes ces informations de typage sont d'ailleurs essentielles à cette analyse et constituent la raison principale de sa faisabilité.

Les attributs se situant à la fin des fichiers de classe sont surtout utilisés pour les fins des logiciels de mise au point de programmes Java. En effet, ils peuvent y emmagasiner des informations qui leur sont précieuses dans ce domaine. Le vérificateur n'est donc pas obligé de vérifier la majorité d'entre eux. Mais l'utilité de ces attributs est souvent sous-estimée. Ils procurent en fait un excellent moyen d'étendre la spécification des fichiers de classe, car il est possible, en respectant certaines règles toutes simples, de définir nos propres attributs et d'y inclure l'information jugée pertinente. Le vérificateur se doit d'ignorer en silence tout attribut qu'il ne reconnaît pas. Ainsi, l'ajout de nouveaux attributs à des fichiers de classe n'empêche pas ceux-ci d'être exécutés par la machine virtuelle.

2.2.2 Analyse de flot de données

L'analyse de flot de données date du début des années 70 et a d'abord été formalisée par Kildall, puis par Kam et Ullman. Aho, Sethi et Ullman traitent en profondeur, dans [Aho et al.86], de cette analyse. Elle est vue comme une technique générale qui permet de scruter statiquement du code pour inférer une grande variété d'informations utiles à plusieurs domaines. Par exemple, à ses débuts, elle était surtout utilisée pour effectuer des optimisations de code. Elle est maintenant adaptée à la vérification formelle de programmes. C'est justement le but visé par le vérificateur de code objet.

L'analyse de flot de données effectuée par le vérificateur simule en fait l'exécution du code d'une méthode en étudiant ses effets sur la pile d'exécution, aussi appelée pile d'opérandes, et les registres. Elle garantit alors que le code de toutes les méthodes de toutes les classes incorporées à la machine virtuelle vérifie les propriétés suivantes : la sûreté de la pile d'opérandes et des registres, la sûreté des types, la sûreté des objets et la sûreté du flot de contrôle. Chacune de ces propriétés sont maintenant étudiées en détail.

2.2.2.1 Pile d'opérandes et registres

À chaque fois qu'une méthode est invoquée, la machine virtuelle lui attribue une pile d'exécution, ou pile d'opérandes, et un certain nombre de registres qui lui appartiennent entièrement. Ainsi, les méthodes ne partagent pas une pile commune ou certains registres comme c'est souvent le cas dans le langage machine de plusieurs plates-formes, comme Windows par exemple. Cette particularité assure une meilleure sécurité, car il devient alors impossible de se servir de la pile ou des registres pour effectuer différentes malices ; l'exécution d'une méthode est donc totalement indépendante des autres méthodes.

À l'endroit où on retrouve le code des méthodes dans le fichier de classe se trouvent aussi des informations très pratiques, générées par le compilateur, concernant la pile d'opérandes et les registres ; on parle du nombre maximum d'éléments que la pile de la méthode peut contenir ainsi que du nombre maximum de registres que la méthode utilise. Ces informations permettent à la MVJ d'allouer l'espace mémoire requis pour l'invocation d'une méthode et à l'analyse de flot de données de s'assurer que la pile d'opérandes ne débordera jamais, que ce soit vers le haut ou vers le bas, en cours d'exécution de la méthode et que les accès aux registres se feront toujours sur des registres valides. C'est ce qui est appelé la sûreté de la pile d'opérandes et des registres.

Étant donné que l'analyse de flot de données assure définitivement cette propriété avant l'utilisation de n'importe quel fichier de classe, la machine virtuelle n'a plus à se soucier de ces problèmes potentiels de débordements de pile et d'utilisation de registres inexistants lors de l'exécution d'une méthode.

2.2.2.2 Types

Une méthode connue depuis un certain temps pour éviter beaucoup de problèmes à l'exécution d'un programme est de s'assurer que les types des valeurs utilisées tout le long de son exécution sont adéquats par rapport aux opérations effectuées sur ces valeurs. De cette façon, il est garanti que ces valeurs sur lesquelles le programme travaille ont un minimum de sens par rapport aux opérations réalisées. Cette propriété est appelée la *sûreté des types*.

Cette propriété est certainement celle qui est au centre de l'analyse de flot de données du vérificateur de code objet. En effet, tout en simulant l'exécution de chaque instruction contenue dans une méthode, l'analyse de flot de données vérifie que les types des valeurs que ces instructions utilisent sont appropriés. Elle doit donc garder une trace des types de toutes les valeurs emmagasinées dans la pile et dans les registres de la méthode. Il faut bien voir que Java a été pensé de façon à rendre cette tâche possible. Par exemple, les instructions du langage machine de la machine virtuelle ne comportent pas d'ambiguïtés quant aux types des valeurs qu'elles doivent utiliser en entrée et celles qu'elles produisent en sortie. L'algorithme de l'analyse de flot de données prend en compte toutes les différentes exécutions possibles. C'est ce qui assure la sûreté des types.

Jetons maintenant un coup d'oeil sur une version grandement simplifiée mais correcte de cet algorithme de l'analyse de flot de données. Notons au préalable qu'un drapeau nommé *changée* est associé à chaque instruction d'une méthode et signale que cette instruction doit être analysée de nouveau par l'algorithme. Avant d'entrer dans l'algorithme présenté ci-dessous, le drapeau *changée* de la première instruction de la méthode est levé. La pile d'opérandes, qui contiendra les types des valeurs empilées, associée à cette instruction est vide. Ses premiers registres contiennent les types des paramètres de la méthode ; tous les autres contiennent un type modélisant une valeur inutilisable, car ils ne sont pas encore initialisés. Toutes les instructions suivantes de la méthode ont leur drapeau *changée* abaissé et ne possèdent pas encore d'informations concernant les types. Voici l'algorithme en question :

1. Trouver une instruction qui a son drapeau *changée* levé. Si aucune instruction ne répond à ce critère, la méthode est acceptée et donc certifiée. Sinon, le drapeau de l'instruction choisie est abaissé et on passe à l'étape 2.

2. Simuler les effets de cette instruction sur la pile et les registres.

    a. Si l'instruction utilise des valeurs prises sur la pile, il faut s'assurer que la pile contient assez de valeurs et que le type des valeurs utilisées est approprié. Si ce n'est pas le cas, la méthode est refusée.

    b. Si l'instruction utilise des valeurs prises dans des registres, il faut s'assurer qu'ils existent et que le type des valeurs qu'ils contiennent est également approprié. Encore une fois, si ce n'est pas le cas, la méthode est refusée.

    c. Si l'instruction empile des valeurs, il faut en fait empiler leur type en s'assurant qu'il reste toujours assez de place dans la pile. Si au contraire la pile déborde vers le haut, il faut refuser la méthode.

    d. Si l'instruction emmagasine des valeurs dans des registres, il faut y placer le type de ces valeurs tout en vérifiant que ces registres existent. Si ce n'est pas le cas, la méthode est refusée.

3. Déterminer les instructions qui suivent l'instruction en cours d'analyse. Ces instructions peuvent être :

    a. L'*instruction suivante*, si l'instruction courante n'est pas *goto*, *return* ou *throw*. Si l'instruction courante pousse à sortir du corps de la méthode, il faut rejeter cette méthode.

    b. *L'instruction cible d'une instruction de branchement conditionnel ou inconditionnel*.

    c. *Toutes les premières instructions des gestionnaires d'exceptions couvrant l'instruction courante*.

4. Fusionner les informations concernant les types de l'instruction courante avec les informations associées à chaque instruction suivante. Dans le cas où l'instruction suivante est la première instruction d'un gestionnaire d'exceptions, la pile doit être vidée et il faut ensuite empiler le type de l'exception qu'il peut gérer.

    a. Si c'est la première fois que l'instruction suivante sélectionnée est visitée, la fusion n'a pas vraiment lieu, car cette instruction n'est associée à aucune information concernant les types. Les informations concernant les types de l'instruction courante sont alors simplement copiées vers cette instruction suivante. Le drapeau *changée* de cette instruction suivante est également levé.

b. Si au contraire l'instruction suivante sélectionnée a déjà été visitée, la fusion s'effectue réellement. Si le résultat de la fusion est différent des informations concernant les types déjà présentes au niveau de l'instruction suivante, le drapeau *changée* de cette instruction est levé.

5. Boucler à l'étape 1.

Pour des raisons de concision et d'adéquation à la portée de ce document, aucun exemple illustrant l'exécution de cet algorithme ne sera présenté.

On remarque cependant qu'ainsi définie, l'analyse de flot de données assure beaucoup plus que la sûreté des types. En effet, elle assure également la sûreté du flot de contrôle, en vérifiant que tous les successeurs de chaque instruction sont effectivement des instructions appartenant à la méthode courante, et ce en prenant en compte les gestionnaires d'exceptions (on verra un peu plus loin que les sous-routines causent par contre des problèmes pour l'assurance de cette propriété). On voit également plus en détail, en étudiant cet algorithme, de quelle façon la sûreté de la pile et des registres est garantie.

Les sous-routines ne sont pas les seules à causer des problèmes à l'analyse de flot de données. En effet, la création et l'initialisation des objets sont des points qui demandent également une attention particulière.

2.2.2.3 Initialisation des objets

La création d'un objet est un processus divisé en deux étapes principales au niveau de la MVJ : l'allocation de la mémoire nécessaire pour emmagasiner l'état de l'objet est suivie de son initialisation. Ainsi, le code suivant écrit en Java :

```
new UneClasse(a, b);
```

est compilé en code objet d'une façon similaire à celle-ci :

```
# alloue l'espace mémoire pour UneClasse et empile la référence
new <UneClasse>
# duplique sur la pile la référence retournée par new
dup
<empiler les paramètres>
# initialise l'espace mémoire de UneClasse
invokespecial UneClasse.<init>
```

Ce code laisse une référence vers la mémoire allouée et initialisée de l'objet sur la pile.

On remarque, en étudiant ce segment de code objet, qu'il y a une période de temps où une référence vers un espace mémoire non initialisé existe sur la pile. Cette référence peut même être dupliquée sur la pile ou copiée dans des registres. Les paramètres du constructeur (méthode nommée `<init>`) doivent également être empilés

avant de pouvoir l'appeler. Il est donc primordial d'empêcher le code d'utiliser cette référence, en appelant une méthode, par exemple, avant qu'il ait correctement initialisé l'espace mémoire vers lequel elle pointe en appelant un constructeur approprié. Si cette contrainte n'existait pas et qu'il était possible d'exécuter du code semblable à ceci :

```
new <UneClasse>
<empiler les paramètres>
invokevirtual UneClasse.uneMethode(a, b, c)
```

il serait alors possible d'appeler une méthode sur un objet qui n'a pas vraiment été construit, car aucun de ses constructeurs n'aurait été appelé. Ce serait contraire aux principes de l'orienté-objet. C'est donc ce qui est appelé la sûreté des objets.

La solution réside dans le fait que l'instruction *new* empile une référence ayant un type spécial qui représente un objet non initialisé. Les instructions *aload* et *astore*, déplaçant respectivement une référence d'un registre vers la pile et de la pile vers un registre, ainsi que *swap* et les variantes de *pop* et de *dup* peuvent toutes manipuler ce type spécial. Par contre, toutes les autres instructions manipulant des références n'ont pas le droit d'utiliser des références ayant ce type spécial. Ainsi, la référence spéciale non initialisée retournée par l'instruction *new* peut être copiée de différentes façons et à différents endroits mais ne peut pas être utilisée, par exemple, pour l'invocation de méthodes à l'aide de l'instruction *invokevirtual*. C'est donc seulement au retour de l'invocation d'un constructeur approprié que le type de la référence passera de non initialisé à un type représentant le nom de la classe de l'objet, signifiant alors que l'objet pointé par la référence est correctement initialisé. C'est l'analyse de flot de données qui doit détecter l'invocation du constructeur et son retour et qui doit alors modifier le type de la référence.

Il faut noter que le problème se complexifie grandement lorsque plusieurs références pointant vers des objets non initialisés différents sont simultanément présentes sur la pile comme dans l'exemple en Java ci-dessous :

```
new BufferedReader(new InputStreamReader(System.in));
```

compilé en un code objet ressemblant à ceci :

```
# alloue l'espace mémoire pour BufferedReader
new <BufferedReader>
dup
# alloue l'espace mémoire pour InputStreamReader
new <InputStreamReader>
dup
# empile le contenu de l'attribut System.in
getstatic System.in
# initialise InputStreamReader
invokespecial InputStreamReader.<init>
# initialise BufferedReader
invokespecial BufferedReader.<init>
```

Il peut même y avoir simultanément sur la pile plusieurs références différentes pointant sur des objets non initialisés du même type. On a donc en Java :

```
new URL(new URL("http", "myhost", 8000, "/dir1/dir2/page.html"),
"../index.html");
```

et en code objet :

```
# alloue l'espace mémoire pour l'URL externe
new <URL>
dup
# alloue l'espace mémoire pour l'URL interne
new <URL>
dup
# empile les paramètres pour l'URL interne
ldc "http"
ldc "myhost"
sipush 8000
ldc "/dir1/dir2/page.html"
# initialise l'URL interne
invokespecial URL.<init>
# empile le deuxième paramètre pour l'URL externe
ldc "../index.html"
# initialise l'URL externe
invokespecial URL.<init>
```

L'analyse de flot de données est donc passablement complexifiée par cette sûreté des objets. Encore une fois, pour des raisons de concision et d'adéquation à la portée de ce document, on ne va pas plus loin dans l'étude de ce problème.

2.2.2.4 Gestion des sous-routines

Il a déjà été mentionné que les sous-routines posent un problème à la sûreté du flot de contrôle. C'est vrai, mais à quoi servent-elles ?

Le langage Java inclut un mécanisme appelé *finally*. Considérons par exemple le code Java suivant :

```
try
{
  ouvrirRobinet();
  arroserPelouse();
}
finally
{
  fermerRobinet();
}
```

Le langage Java garantit que le robinet sera fermé, peu importe de quelle manière la section de code englobée par le *try* se termine. Ainsi, même si une exception survient pendant l'ouverture du robinet ou pendant l'arrosage de la pelouse, le robinet sera fermé. Ce mécanisme est bien pratique dans ce cas-ci pour prévenir des dégâts d'eau désastreux !

La section de code englobée par le *finally* est en fait compilée sous la forme d'une sous-routine. Cette sous-routine n'est qu'un segment de code objet inclus dans le code de la méthode contenant le *finally* et ne peut être appelée que par cette méthode.

Voilà donc une nouvelle entité du point de vue du flot de contrôle : la *sous-routine*. Après avoir été appelée, elle devra tôt ou tard se terminer et revenir à l'instruction qui suit l'instruction qui a effectué l'appel. C'est précisément ce retour de sous-routine que l'analyse de flot de données doit contrôler. En effet, comme l'instruction de retour de sous-routine *ret* prend l'adresse de retour dans un registre dont le numéro est spécifié en argument et que plusieurs adresses de retour différentes peuvent se retrouver simultanément dans différents registres (à cause d'appels de sous-routines imbriqués, par exemple), l'analyse de flot de données doit s'assurer que cette instruction ne sera pas utilisée afin de brancher n'importe où dans le code de la méthode. Elle doit en effet vérifier que les appels et les retours de sous-routines se font de façon standard, c'est-à-dire que la dernière sous-routine appelée est la première à terminer et qu'elle retourne bien vers le code appelant (elle pourrait en fait retourner vers n'importe quelle sous-routine qui est dans sa chaîne d'appels).

Essentiellement, l'analyse de flot de données doit savoir, pour n'importe quelle sous-routine *s*, vers quelles sous-routines cette sous-routine *s* peut retourner. Pour ce faire, une solution est d'associer un ensemble appelé *retournables* à chaque instruction d'une sous-routine de façon à y placer les adresses vers lesquelles cette sous-routine peut retourner. Le contenu de cet ensemble est alors propagé et augmenté d'un appel de sous-routine à l'autre. Ceci fait en sorte que chaque instruction d'une sous-routine *s* possède un ensemble *retournables* qui contient les adresses de retour des sous-routines présentes dans la chaîne d'appels de *s*.

Les sous-routines sont également problématiques pour l'algorithme de l'analyse de flot de données en tant que tel. Prenons par exemple une instruction qui peut être atteinte en parcourant divers chemins d'exécution différents. Nommons cette instruction *i*. Supposons maintenant qu'un registre *r* donné contient des valeurs ayant des types incompatibles d'un chemin à l'autre. Lorsque l'analyse de flot de données aura terminé

de simuler l'exécution des différents chemins qui mènent à *i*, le type de son registre *r* modélisera une valeur inutilisable. C'est exactement ce qui se passe avec les appels de sous-routines. Pouvant être appelées à partir de plusieurs endroits dans le code de la méthode, leur instruction de retour *ret* est atteignable en parcourant des chemins d'exécution différents. Cette même instruction *ret* a donc également plusieurs successeurs possibles : chaque instruction qui suit celles appelant la sous-routine. Ces successeurs verront donc certains de leurs registres contenir un type caractérisant une valeur inutilisable. Le problème est que les valeurs qui y étaient emmagasinées avant d'appeler la sous-routine doivent bien souvent être utilisées après l'avoir appelée. L'algorithme de flot de données utilisé tel que présenté plus tôt invalidera donc ces registres même si la sous-routine appelée n'y touche pas. Cela rend les sous-routines à peu de choses près inutiles.

La solution, étant loin d'être simple, a pour idée principale de modifier l'algorithme de l'analyse de flot de données afin de faire en sorte qu'il ne traite pas les registres non modifiés par les sous-routines. On dit donc, dans le jargon du vérificateur de code objet, que les sous-routines doivent être polymorphes sur les registres qu'elles n'utilisent pas. En supportant ce polymorphisme, on redonne l'utilité recherchée aux sous-routines. On ne s'attarde pas plus sur ce problème dans ce document.

Passons plutôt à l'explication de la sécurité de plus haut niveau de Java.

2.3 Gestionnaire de sécurité Java

Java utilise le concept de permission pour gérer la sécurité. Une *permission* représente la possibilité d'accéder à une ressource. En Java, une *politique de sécurité* est un ensemble de permissions définies dans un fichier. Par défaut, la politique de sécurité des applications inclut toutes les permissions, c'est-à-dire qu'elles peuvent accéder à toutes les ressources, et celle des applets n'inclut presque pas de permissions, c'est-à-dire que leurs accès aux ressources sont très limités. En fait, elles ne peuvent pas accéder au système de fichiers. Elles peuvent par contre recevoir et envoyer de l'information au serveur d'où elles proviennent.

Java ne tient pas compte de ces politiques de sécurité si un gestionnaire de la sécurité n'est pas installé au démarrage de la MVJ. En effet, la gestion de la sécurité se fait totalement de façon dynamique dans Java. Les appels aux ressources dans la librairie standard de Java ont recours au gestionnaire de la sécurité, qui consulte alors la politique de sécurité du système pour déterminer si un appel donné est légal ou non. Si aucun gestionnaire de la sécurité n'est installé, la vérification ne se fait pas et l'appel à la ressource est accepté. Par défaut, il y a donc un gestionnaire de la sécurité installé dans les MVJ qui résident dans les navigateurs. Les politiques de sécurité par défaut à ce niveau spécifient les permissions nécessaires minimales pour que les applets puissent fonctionner mais soient limitées au maximum afin d'être potentiellement le moins malicieuses possible.

Pour les applications par contre, la politique de sécurité définie par défaut suite à l'installation du JSDK (de l'anglais « Java Software Development Kit ») 1.2 ou 1.3

spécifie que tout est permis. Ainsi, si aucun changement n'est effectué à cette politique et si aucun gestionnaire de la sécurité n'est installé au démarrage de la MVJ, la MVJ garantit que les applications sont sûres, par l'utilisation du vérificateur de code objet Java, mais pas sécuritaires. On doit modifier la politique par défaut et installer un gestionnaire de la sécurité au démarrage de la MVJ pour restreindre les applications exécutées.

Une permission peut être accordée à toute la planète ou à un acteur (ou *principal*) donné et/ou à un URL donné. Évidemment, cela s'applique beaucoup mieux aux applets, car elles proviennent d'un URL donné et peuvent être signées (par la cryptographie) par un acteur. Mais étant donné que les fichiers JAR (de l'anglais « Java Archive ») peuvent aussi être signés, il est également possible de définir des permissions pour un acteur donné pour les applications.

Une permission possède généralement un type, un sous-type et une action. Par exemple, une permission de type `FilePermission` possède un sous-type décrivant le(s) fichier(s) sur le(s)quel(s) la permission est effective et une action qui décrit quelle est l'opération permise sur ce(s) fichier(s). Par exemple, cette politique de sécurité définit une seule permission pour la planète :

```
grant
{
  permission java.io.FilePermission "*.tmp", "read";
};
```

Cette permission permet à tout le monde de lire (action) n'importe quel fichier ayant l'extension *.tmp* (sous-type) sur le système de fichiers du client.

Voici une autre politique :

```
grant codeBase "http://java.sun.com/*", signedBy "JamesGosling"
{
  permission java.io.FilePermission "/tmp/*", "read";
  permission java.io.SocketPermission "*", "connect";
};
```

Cette politique permettrait à tout code provenant du site Web `java.sun.com` et signé par `James Gosling` de lire tous les fichiers contenus dans notre répertoire `tmp` et de se connecter à n'importe quel serveur sur la planète. Pour que Java puisse savoir si le code est signé par `James Gosling`, une base de données qui associe des noms de principaux à des clés publiques doit être gérée. Java inclut tout ce qu'il faut pour que cela soit possible et facile.

Cette dernière politique pourrait être ajoutée à la première politique présentée pour créer une troisième politique, qui donnerait les permissions déjà définies plus haut à toute la planète et les permissions qui viennent tout juste d'être définies, au code signé par `James Gosling` et provenant du site Web `java.sun.com`.

Maintenant que les éléments centraux propres à la sécurité de Java ont été présentés, passons à l'étude du premier article.

3 Premier article – Bytecode Model Checking: An Experimental Analysis

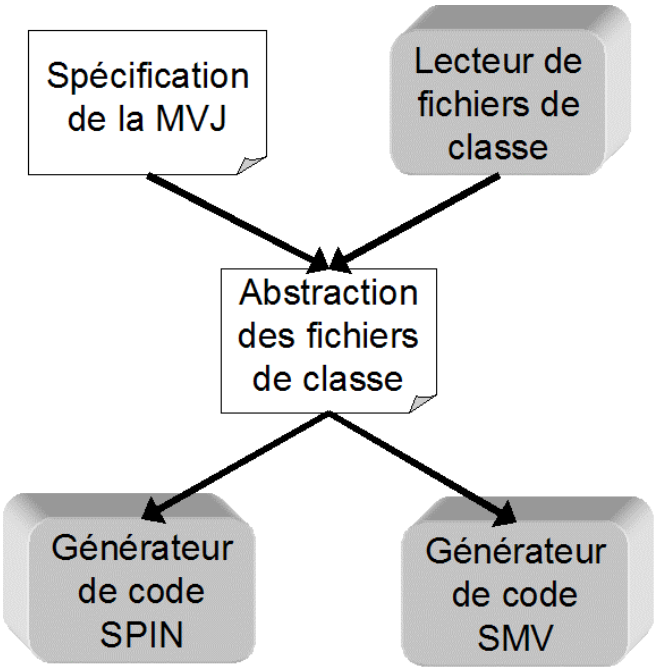
Ce premier article présente une approche visant à remplacer l'analyse de flot de données effectuée par le vérificateur de code objet de la machine virtuelle Java. Cette approche est basée sur le *model checking*[3]. Bien que les algorithmes de model checking ont en général une complexité exponentielle en pire cas, il est possible d'obtenir une complexité polynomiale en moyenne sur des fichiers de classe *correctement typés* en utilisant des algorithmes de model checking à états explicites (de l'anglais « explicit state, on-the-fly model checker »). La complexité de cette nouvelle approche est alors du même ordre que celle de l'analyse de flot de données. Cependant, quelques bémols seront émis un peu plus loin dans cette section quant à la complexité des algorithmes de model checking pour l'analyse de fichiers de classe *incorrectement typés*. De plus, telle que présentée dans ce premier article, cette approche ne vérifie que les propriétés de sûreté scrutées par l'analyse de flot de données du vérificateur. Les autres propriétés, assurées notamment pour la structure des fichiers de classe, sont supposées être vérifiées par d'autres techniques.

Étudions maintenant les détails de l'architecture de cette approche.

3.1 Architecture

La figure qui suit illustre l'architecture générale de l'approche présentée par l'article étudié dans cette section :

[3] Les termes « model checking » et « model checker » ne sont pas traduits de l'anglais et ne sont pas mis entre guillemets pour alléger le texte.



Le système développé prend donc en entrée des fichiers de classe Java et une spécification d'une abstraction de la machine virtuelle Java (détaillée à la sous-section 3.3). Ces informations sont utilisées afin de générer une représentation intermédiaire qui consiste en un système de transitions (représentant les abstractions des méthodes des classes fournies en entrée et détaillé à la sous-section 3.3.1.1) et une spécification des propriétés de sûreté formalisant les conditions suffisantes pour assurer la sûreté des types (détaillée à la sous-section 3.3.1.2). Cette représentation intermédiaire est finalement traduite en des langages permettant de vérifier que le système de transitions respecte les propriétés de sûreté à l'aide des model checkers SPIN et SMV. Ces dernières traductions sont un peu moins intéressantes dans le cadre du présent document et ne seront donc pas détaillées. Le lecteur intéressé est référé à [Basin et al.02].

Avant d'entrer dans les détails des différents modules de cette architecture, les motivations ayant poussé les auteurs à développer une telle approche sont présentées.

3.2 Motivations

Trois motivations principales ont poussé les auteurs à faire de la recherche afin de développer cette nouvelle approche :

1. *Utiliser les méthodes formelles pour améliorer la sécurité des processus de vérification et d'exécution de la machine virtuelle Java*. Ceci permet de fournir une documentation plus précise et moins ambiguë des mécanismes internes de la machine virtuelle. En effet, la documentation fournie par Sun Microsystems Inc. est très incomplète et souvent assez peu précise. De plus, l'utilisation d'un model checker permet d'éliminer plusieurs erreurs d'implantation.

2. *Déterminer si le model checking est applicable à une plate-forme telle que Java*. Cet objectif n'est pas trivial. La vérification conventionnelle, utilisant une analyse de flot de données, possède une complexité polynomiale. Cependant, les algorithmes de model checking possèdent une complexité exponentielle en pire cas. Les auteurs ont donc dû démontrer que pour des fichiers de classe *correctement typés*, le model checking a une complexité polynomiale, comparable à l'analyse de flot de données. La raison qui explique ce phénomène est que malgré le nombre exponentiel d'états, pour du code *correctement typé*, seulement un nombre polynomial d'états sont atteignables. Ainsi, il est important d'utiliser un algorithme de model checking à états explicites (tel que celui utilisé par SPIN), car les algorithmes de ce type construisent l'espace d'états de façon incrémentale, c'est-à-dire sur demande, au cours de la vérification, plutôt que de le construire en totalité, avant de débuter la vérification.

   Les résultats obtenus par les auteurs suggèrent que l'utilisation du model checking pour la vérification des fichiers de classe Java est applicable pour les domaines où les besoins en temps et en espace sont moins importants que la correction et l'extensibilité du système. Par exemple, dans le monde des cartes à puces Java, la vérification ne peut pas être faite sur la carte, car les contraintes en espace mémoire sont trop importantes. Dans cette situation, la vérification pourrait être effectuée par un model checker, avant d'écrire le code sur la puce de la carte.

3. La troisième motivation est indépendante des deux premières. En effet, les auteurs ont choisi la plate-forme Java en partie parce qu'*il y a d'innombrables exemples de taille réaliste et facilement accessibles pour vérifier leurs hypothèses*.

Analysons maintenant de façon plus approfondie le système développé par les auteurs.

3.3 Système

Comme présenté à la section 3.1, le système en question a une architecture assez simple. En fait, le seul module particulièrement intéressant est le module central, c'est-à-dire la représentation intermédiaire ou l'*abstraction des fichiers de classe*. En effet, la *spécification de la MVJ*, quoique présentée comme une entité distincte, constitue plutôt une notation ou un formalisme utilisé dans l'abstraction des fichiers de classe. Ainsi, la MVJ n'est pas réellement totalement spécifiée dans un seul document. Ce module dans l'architecture représente plutôt un formalisme vu à la section 3.3.1, lorsque l'abstraction des fichiers de classe sera couverte.

De même, le module de *lecture de fichiers de classe* n'est pas particulièrement intéressant dans le cadre de ce document, car il ne s'agit après tout que d'un désassembleur, qui est d'ailleurs assez facile à implanter pour les fichiers de classe Java.

Passons alors directement au module de l'abstraction des fichiers de classe.

3.3.1 Abstraction des fichiers de classe

L'abstraction des fichiers de classe comprend deux entités : un système de transitions et une spécification des propriétés de sûreté.

3.3.1.1 Système de transitions

Les fichiers de classe Java sont abstraits méthode par méthode. Une méthode *M* est abstraite par un système de transitions (fini) défini par un triplet $\langle Q, q_0, \Delta \rangle$. L'ensemble $Q \subseteq \mathbb{N}$ x (*T* pile) x (*T* tableau) d'états contient des triplets *(cp, pile, regs)* qui consistent en un compteur de points de programme, la pile d'opérandes et le tableau de registres de la méthode *M*. L'ensemble *T* de types contient les types primitifs et les types de référence de la machine virtuelle. Cet ensemble contient aussi un type *NULL* qui représente le type *null* polymorphe de la machine virtuelle ainsi que les types *ADR* et *UNDEF* qui représentent respectivement le type des adresses de retour utilisé par l'instruction *ret* de la machine virtuelle et le type d'une référence non initialisée. Ainsi :

$$prim = \{INT, FLOAT, LDOUBLE, HDOUBLE, LLONG, HLONG\}$$
$$ref = \{REF(cn) \mid cn \in noms\_de\_classe\} \cup \{NULL\}$$
$$adr = \{ADR(i) \mid i \in \mathbb{N}\}$$
$$T = \{UNDEF\} \cup prim \cup ref \cup adr$$

Un sous-ensemble fini de l'ensemble infini *T* peut être calculé pour chaque méthode, car pour une méthode donnée, seulement un sous-ensemble fini de types dans *T* peut être présent. Ce calcul est fait à partir de la signature de la méthode et de son corps. La signature d'une méthode peut être définie comme suit :

$$S = \langle M, type\_retour, [type\_arg_1, ..., type\_arg_n] \rangle$$

où *M* spécifie le nom de la méthode, *type_retour* désigne le type de retour de la méthode et [$type\_arg_1$, ..., $type\_arg_n$] définit le type des arguments de la méthode, s'il y a lieu.

Le corps de la méthode est exprimé comme une liste *ins* d'instructions de code objet Java (aussi appelées *instructions de la machine virtuelle Java*). Chaque instruction d'une méthode introduit donc potentiellement un nouveau type dans son ensemble *T*. Par exemple, l'instruction *imul* (multiplication sur les entiers) introduit le type *INT* s'il n'est pas déjà présent dans *T*.

Ainsi, une définition opérationnelle de l'ensemble *T* pourrait être :

$$T = \{REF(C), type\_retour, type\_arg_1, ..., type\_arg_n\} \cup \bigcup_{p \in \{0,...,|ins|-1\}} types\_de\_ins(ins_p)$$

où *C* est le nom de la classe de la méthode.

L'état initial q_0 du système de transitions $\langle Q, q_0, \Delta \rangle$ pour une méthode *M* possédant une signature *S* et appartenant à une classe *C* est calculé de la façon suivante :

$$q_0 = (0, \text{Vide}, regs), \text{où} \begin{cases} regs[0] = REF(C) \\ regs[1] = type\_arg_1, ..., regs[n] = type\_arg_n \\ regs[n+1] = UNDEF, ..., regs[maxloc] = UNDEF \end{cases}$$

Ainsi, le compteur de points de programme est initialisé à 0, la pile d'opérandes est vide et le tableau de registres *regs* possède une première entrée initialisée à la référence *this*, qui pointe sur la classe courante, suivi des arguments de la méthode. Si le tableau de registres est plus grand que le nombre d'arguments de la méthode, les espaces vides sont initialisés à *UNDEF*. La variable *maxloc* contient le nombre maximum de registres que la méthode *M* peut utiliser. Ce nombre est fourni pour chaque méthode dans les fichiers de classe.

La relation de transitions Δ du système de transitions $\langle Q, q_0, \Delta \rangle$ d'une méthode *M* est définie par les instructions de code objet Java de la méthode *M*. Cette relation de transitions correspond en fait aux fonctions de transfert de l'analyse de flot de données. Cette relation prend en entrée un triplet *(cp, pile, regs)* et le modifie en fonction de l'instruction en cours. Par exemple, pour l'instruction *imul*, la relation est définie comme suit :

$$(cp, pile, regs) \mapsto \{(cp+1, INT.pop(pop(pile)), regs)\}$$

Ceci nous amène à la spécification des propriétés de sûreté.

3.3.1.2 Spécification des propriétés de sûreté

Les propriétés de sûreté sont formalisées sous forme de prédicats sur les états du système de transitions. Ces propriétés doivent être satisfaites de façon globale, pour chaque exécution possible du système de transitions (qui abstrait en fait une méthode). Deux propriétés sont définies par les auteurs :

1. *La pile d'opérandes ne doit pas déborder vers le haut*. Cette propriété est formalisée comme suit :

   $size(pile) \leq maxstack$

   où *maxstack* est précisé dans le fichier de classe.

2. *Chaque instruction doit toujours opérer sur des données d'un type approprié*. Pour vérifier cette propriété dans le contexte d'un langage orienté-objet comme Java, une relation $\sqsubseteq_\Gamma$ de sous-typage doit être définie pour chaque programme Γ vérifié. Pour un programme Γ donné, la relation $\sqsubseteq_\Gamma$ est finie, car un programme

ne peut définir ou utiliser qu'un nombre fini de types. Le typage des instructions de la machine virtuelle (ou, dans un sens opérationnel, la machine virtuelle elle-même) est spécifié en utilisant cette relation $\sqsubseteq_\Gamma$. Par exemple, le typage de l'instruction *imul* est formalisé de la façon suivante :

$$imul \mapsto \left(top(pile) \equiv INT\right) \wedge \left(top(pop(pile)) \equiv INT\right)$$

Cette propriété signifie simplement que les deux premiers éléments de la pile doivent être du type *INT*. Dans le cas d'une instruction manipulant des références, la relation $\sqsubseteq_\Gamma$ est utilisée pour déterminer quels sont les types compatibles, du point de vue de l'héritage, à un type de référence donné.

Ainsi, la propriété de sûreté globale pour une méthode est formalisée comme suit :

$$\left(size(pile) \leq maxstack\right) \wedge \bigwedge_{p \in \{0,\ldots,|ins|-1\}} \left(cp = p \Rightarrow typage\_de\_ins(ins_p)\right)$$

Finalement, les systèmes de transitions ainsi que cette propriété de sûreté sont traduits dans des langages appropriés pour être vérifiés automatiquement par les model checkers SPIN[4] et SMV. Ces détails sont d'ordre technique et ne nous intéressent pas dans le cadre du présent document. Encore une fois, le lecteur intéressé est référé à [Basin et al.02].

Les auteurs ont effectué plusieurs tests sur leur système et ils ont présenté divers graphiques, ce qui a permis de faire l'analyse suivante.

3.4 Analyse

Dans cette sous-section, la complexité de la vérification conventionnelle, utilisant une analyse de flot de données, est comparée à la complexité de l'approche utilisant le model checking. Supposons que $|ins|$ est le nombre de points de programme (dans notre cas, les points de programme correspondent aux instructions du code objet), $|T|$ est le nombre de types, *maxpile* est la hauteur maximale de la pile d'opérandes et *maxregs* est le nombre de registres. Ces valeurs sont toutes définies pour une méthode donnée. La taille de l'espace d'états de la vérification conventionnelle serait :

$$|ins| \cdot \left(2^{|T|}\right)^{maxpile+maxregs}$$

puisque chaque point de programme est associé à un ensemble de types (dans le cas où la sûreté du typage des interfaces est considérée, c'est-à-dire qu'il y a présence d'héritage

[4] Pour SPIN, le langage approprié est Promela.

multiple) pour chaque élément de la pile et chaque registre. Malgré cet espace d'états exponentiel en théorie, en pratique, seulement un nombre linéaire d'états doivent être visités. Ceci est garanti par les contraintes imposées sur le code objet, en particulier le fait que deux sous-routines ne peuvent pas être terminées par la même instruction *ret*. Ainsi, l'analyse de flot de données ne fait qu'itérer sur les points de programme en calculant le « supremum » des types lorsqu'elle visite plusieurs fois le même point de programme. En pratique, le point fixe est donc calculé après :

$$O\left(|ins| \cdot |T| \cdot (maxpile + maxregs)\right)$$

itérations.

Dans le cas du model checking, la taille de l'espace d'états est :

$$|ins| \cdot |T|^{maxpile+maxregs}$$

Elle est donc aussi exponentielle en théorie. En pratique, la complexité du model checking dépend évidemment de l'algorithme utilisé. Les algorithmes dits symboliques (de l'anglais « symbolic methods ») manipulent une représentation complète de l'espace d'états et peuvent donc avoir une complexité exponentielle pour parvenir à leur fin. Les algorithmes dits explicites (de l'anglais « explicit methods »), calculent cependant l'espace d'états au fur et à mesure que la vérification progresse. Ils sont donc généralement beaucoup plus performants. En fait, en supposant que l'imbrication des sous-routines est *limitée* et que la méthode est *correctement typée*, seulement

$$O\left(|ins|\right)$$

états doivent être visités puisqu'il y a un seul type possible pour chaque point de programme (en abstrayant les sous-types compatibles).

Les deux approches ainsi comparées du point de vue de leur complexité, il est de mise de souligner quelques limites actuelles de l'approche utilisant le model checking.

3.5 Limites actuelles

À la lecture de cet article, il a été possible de mettre en lumière quelques limites à cette approche utilisant le model checking, telle qu'implantée par les auteurs :

- *Quand le code objet est incorrectement typé, le model checking peut ne pas terminer ou utiliser toute la mémoire disponible, et ce peu importe s'il utilise un algorithme symbolique ou explicite*. En effet, les auteurs mentionnent eux-mêmes cette limite. Cependant, ils affirment que ce n'est pas un problème en pratique en argumentant que lorsque trop de ressources sont utilisées, la vérification peut être arrêtée en donnant une réponse conservatrice ou une autre approche peut être utilisée. Cette limite est tout de même gênante. En effet, donner une réponse trop conservatrice, voire même plus conservatrice que ce que donnerait une analyse de

flot de données, rend la vérification plus ou moins utile. De plus, l'utilisation d'une autre approche rendrait l'approche du model checking plus complexe et alors, éliminerait un de ses avantages sur l'analyse de flot de données.

- *Telle qu'implantée par les auteurs, l'approche ne tient pas compte de la sûreté de l'initialisation des objets*. Les auteurs mentionnent aussi ce point en conclusion. Ils évoquent aussi le fait que ce problème ne se résout pas simplement à l'aide de leur approche. En effet, les types à intégrer pour assurer la sûreté de l'initialisation des objets sont assez complexes et leur état n'est pas simple à maintenir en utilisant le model checking. Ainsi, bien que la sûreté du typage des interfaces ainsi que la sûreté des sous-routines (si et seulement si leur niveau d'imbrication n'est pas élevé) soit assez simple à gérer par model checking, l'initialisation des objets est complexe et n'est en fait pas encore implantée. De plus, cette initialisation des objets est relativement simple à implanter par analyse de flot de données. Cette affirmation est aussi vraie pour la sûreté du typage des interfaces. Finalement, les auteurs ne mentionnent pas la nécessité de considérer la gestion des exceptions dans la vérification des fichiers de classe Java (et pourtant, elle est nécessaire). Il est possible qu'ils considèrent que ce problème est résolu par une autre approche mais dans tous les cas, l'ajout de cette gestion complexifierait leur technique. Ainsi, du point de vue de la complexité de l'implantation, les deux approches sont relativement équivalentes, contrairement à ce que les auteurs sous-entendent, car l'analyse de flot de données est également assez complexe à implanter pour Java.

- *L'approche ne permet pas de spécifier certaines propriétés de sûreté assurées par l'analyse de flot de données*. Ce point n'est pas indiqué par les auteurs. Par exemple, la résolution, c'est-à-dire la vérification que les méthodes ou attributs référencés dans une classe existent vraiment et sont bien définis en termes de typage et d'accessibilité, n'est pas formalisable actuellement en utilisant l'approche en question. Une fois de plus, les auteurs supposent peut-être que la résolution est effectuée à l'aide d'une autre technique.

- *Les auteurs ont oublié de définir une propriété de sûreté : l'assurance que la pile d'opérandes ne déborde pas vers le bas*. Cette propriété est aussi importante que sa duale : le non-débordement de la pile vers le haut. Elle pourrait être définie de la façon suivante :

  $size(pile) \geq 0$

3.6 Conclusion du premier article

Pour conclure, cette approche utilisant le model checking pour remplacer l'analyse de flot de données effectuée au niveau du vérificateur de code objet Java est prometteuse. Cependant, il reste encore du travail et de la recherche à faire pour repousser les limites susmentionnées. C'est une fois ce travail effectué qu'il sera possible de déterminer si cette approche est la meilleure.

L'approche détaillée dans cet article ne visait qu'à assurer la sûreté du typage au niveau des fichiers de classe Java. Ceci n'est pas suffisant en pratique pour assurer la sécurité d'un système critique. Le prochain article introduit une nouvelle approche pour assurer une sécurité de plus haut niveau pour les applets et les services Jini de Java.

4 Deuxième article – Mobile Code Security by Java Bytecode Instrumentation

Ce deuxième article présente une approche visant à instrumenter le code objet Java pour permettre d'assurer une meilleure sécurité des applets et des services Jini. Comme présenté à la sous-section 2.3, la plate-forme Java possède déjà une architecture de sécurité lui permettant d'assurer certaines contraintes ou permissions sur les ressources du système hôte. Cependant, ces permissions ne sont pas assez expressives pour définir des politiques de sécurité complexes, tel qu'exigé pour les environnements critiques. En fait, cette nouvelle approche, basée sur l'instrumentation du code objet, permet de détecter et d'arrêter certains types de déni de service, d'assurer l'intégrité et la confidentialité des données critiques du système et même de contrecarrer certains types de « spoofing ». Cette approche est donc plus expressive et plus flexible que les techniques utilisées à ce niveau dans Java.

Arrêtons-nous sur les détails de cette approche.

4.1 Instrumentation

Somme toute, les instrumentations de code objet Java décrites dans cet article sont assez simples à comprendre. Les auteurs ont implanté deux types d'instrumentation, détaillés dans cette sous-section :

1. Instrumentation au niveau d'une classe et

2. instrumentation au niveau d'une méthode.

4.1.1 Instrumentation au niveau d'une classe

L'instrumentation au niveau d'une classe est le plus simple des deux types d'instrumentation. Le principe de base consiste à remplacer, dans le code objet du programme Java à vérifier, toutes les utilisations d'une classe `A` jugée critique par une autre classe `B`, qui est une sous-classe de la classe `A`. Ceci est possible et facile à faire, car `B` est une sous-classe de `A` et donc, partout où `A` est utilisée dans le code objet, `B` peut l'être aussi.

Supposons, par exemple, que la classe `Window` de la librairie standard de Java est jugée critique, car elle permet à un programme Java de créer et d'ouvrir des fenêtres. En effet, cette capacité peut être utilisée à des fins malicieuses en créant et en ouvrant plusieurs dizaines de fenêtres afin d'encombrer le moniteur de l'utilisateur. On aimerait donc être en mesure de compter le nombre de fenêtres qu'un programme Java ouvre afin

d'imposer une certaine limite. Par exemple, on pourrait vouloir limiter le nombre de fenêtres ouvertes à 20. En utilisant l'instrumentation au niveau d'une classe, on peut définir une nouvelle classe, appelée `SafeWindow`, qui hérite de la classe `Window`, et dont la méthode `show`, qui ouvre la fenêtre créée, incrémente une variable qui conserve le nombre de fenêtres ouvertes. Si la variable vaut 20 avant l'incrémentation, on lève une exception afin de signaler la situation non souhaitée. Lorsque la méthode `close` est appelée, la même variable est décrémentée. De cette façon, on s'assure qu'un maximum de 20 fenêtres pourront être ouvertes en même temps.

Il suffit par la suite de commander au système implanté par les auteurs d'instrumenter le code objet des programmes Java en remplaçant les utilisations de la classe `Window` par des utilisations de la classe `SafeWindow` nouvellement implantée. Bien qu'il ne soit pas intéressant de couvrir ces détails techniques dans le cadre de ce document, les changements à faire au niveau des fichiers de classe sont très simples pour ce type d'instrumentation. Il suffit en fait de changer quelques valeurs (`Window` pour `SafeWindow`, etc.) dans le regroupement des constantes des fichiers de classe concernés.

Cependant, comme ce type d'instrumentation implique que l'on doit sous-classer une classe, il ne peut pas s'appliquer aux classes déclarées *final* ou aux interfaces. Pour sécuriser de telles classes ou les interfaces, il faut utiliser l'instrumentation au niveau d'une méthode.

4.1.2 Instrumentation au niveau d'une méthode

L'instrumentation au niveau d'une méthode suit le même principe que l'instrumentation au niveau d'une classe : du code permettant de contrôler l'exécution d'un programme est ajouté à ce programme. Cependant, contrairement à l'instrumentation au niveau d'une classe où le code est ajouté dans une sous-classe nouvellement définie, pour l'instrumentation au niveau d'une méthode, le code est ajouté dans une nouvelle méthode qui remplacera une méthode préexistante dans le programme à surveiller. Ce type d'instrumentation permet d'avoir plus de flexibilité. Par exemple, une classe déclarée *final* peut être surveillée.

Par exemple, supposons que la classe `Window` est déclarée *final* et qu'on ne peut alors pas la sous-classer. Il serait possible d'utiliser l'instrumentation au niveau des méthodes `show` et `close`, en définissant le même code que dans l'exemple de la sous-section 4.1.1 afin de parvenir aux mêmes résultats.

Pour effectuer ce type d'instrumentation, il faut donc fournir au système développé par les auteurs la nouvelle méthode à injecter ainsi que le nom et l'emplacement de la méthode à remplacer dans le programme à surveiller. Cette fois, les changements à faire au niveau des fichiers de classe sont assez complexes, mais débordent toujours du cadre de ce document.

Les auteurs ont testé leur approche sur deux types de programme Java : les applets et les services Jini. La mise en place de leur approche dans ces deux contextes diffère

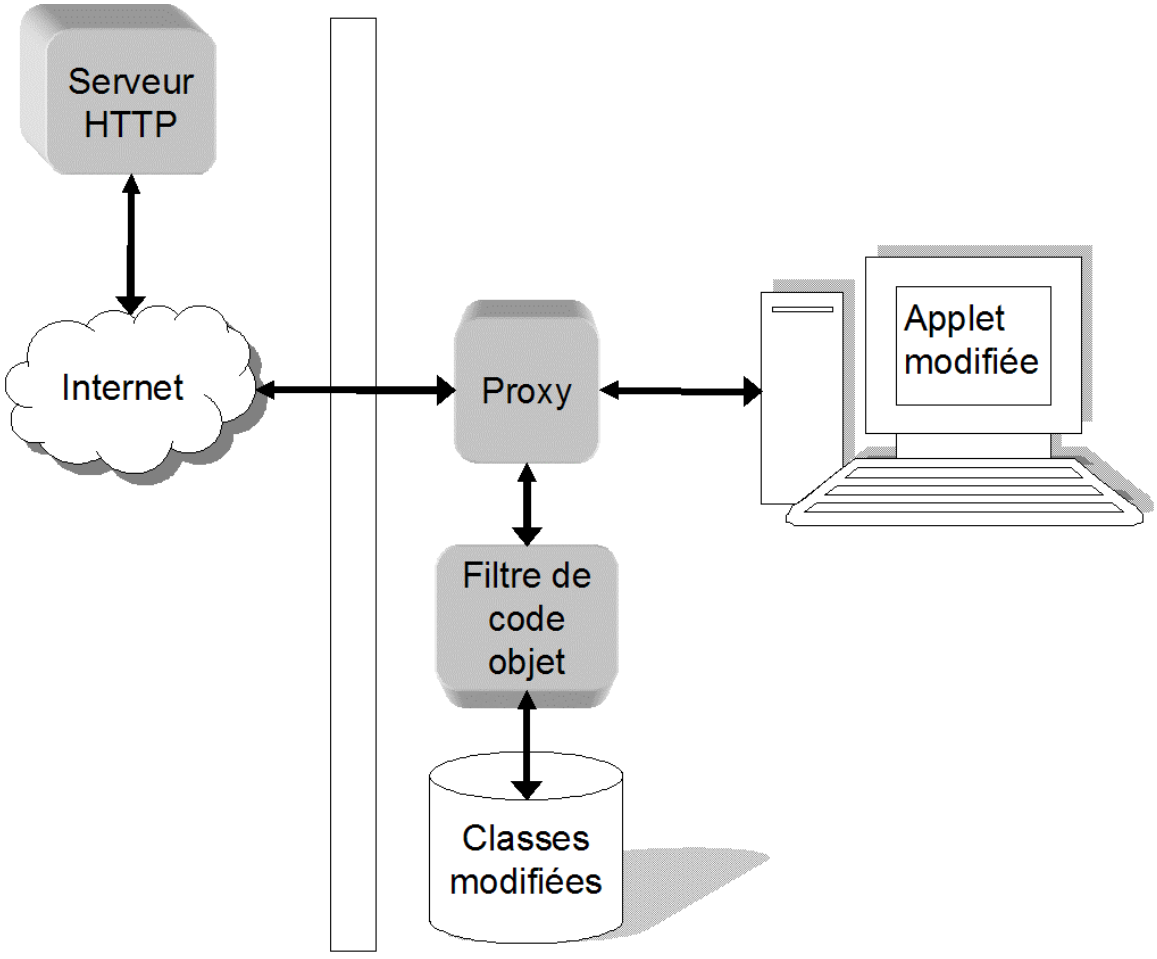
l'une de l'autre. Les détails de ces deux mises en place constituent le sujet de la prochaine sous-section.

4.2 Mise en place de l'approche

Dans cette sous-section, on décrit comment les auteurs ont mis leur approche en place dans le contexte des applets et des services Jini.

4.2.1 Applets

Le code objet des applets est téléchargé en utilisant le protocole HTTP, le même protocole utilisé pour télécharger les pages Web. Ce protocole étant standardisé, il est assez facile d'intégrer l'approche des auteurs à l'exécution des applets. Les auteurs ont utilisé l'architecture suivante pour ce faire :



Cette architecture est basée sur l'utilisation d'un « proxy ». Ce logiciel est interposé entre l'Internet et le client et ne fait qu'acheminer les requêtes HTTP du client vers l'Internet. Cependant, il est très bien placé pour modifier le code objet des applets téléchargées avant qu'elles n'arrivent dans l'ordinateur du client. Ce « proxy » interagit donc avec un filtre de code objet qui effectue l'instrumentation en fonction des ordres

qu'il a reçus. Ce filtre a accès à une base de données dans laquelle reposent les classes et les méthodes modifiées implantées par l'administrateur.

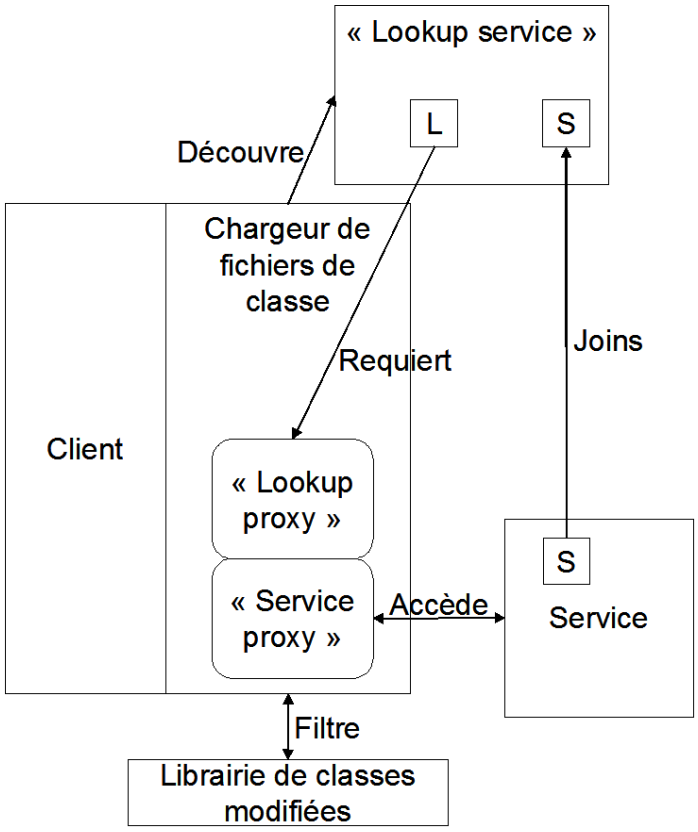
4.2.2 Services Jini

Jini étend la librairie standard de Java d'une seule machine virtuelle à un réseau de machines virtuelles. En général, un système Jini contient des services qui offrent certaines fonctionnalités à tous les membres d'une fédération de machines virtuelles Java. Jini rend donc la programmation répartie beaucoup plus facile à gérer.

Cependant, l'architecture de Jini nécessite que les clients désireux d'utiliser un des services disponibles téléchargent des programmes mobiles appelés « lookup » et « service proxies ». Comme tout programme mobile, ces « proxies » comportent des risques du point de vue de la sécurité. Tout comme les applets, il faut donc contrôler leur exécution.

Cependant, le standard de Jini ne spécifie pas de protocole de base à utiliser pour les communications entre les machines virtuelles. La plupart du temps, RMI (de l'anglais « Remote Method Invocation ») est utilisée mais les implantations de Jini ne sont pas obligées de l'utiliser. Ainsi, il n'est pas possible de faire comme pour les applets et de se placer au niveau du réseau pour intercepter les « lookup » et « service proxies » afin de les instrumenter.

Les auteurs ont donc utilisé l'architecture suivante pour parvenir à leurs fins :



En fait, la solution apportée est simple : le chargeur de fichiers de classe du client est modifié afin d'instrumenter les « lookup » et « service proxies » reçus du « lookup service ». En effet, les services enregistrent leur « service proxy » auprès du « lookup service ». Lorsqu'un client requiert les services disponibles auprès du « lookup service », il reçoit le « lookup proxy » du « lookup service ». De même, lorsqu'il demande à utiliser un service en particulier, il reçoit le « service proxy » de ce service. Ces « lookup » et « service proxies » sont d'abord filtrés par le chargeur de fichiers de classe afin de les instrumenter pour contrôler leur exécution selon le code qui leur est ajouté.

4.3 Limites actuelles

À la lecture de cet article, il a toutefois été possible de mettre en lumière quelques limites à cette approche utilisant l'instrumentation, telle qu'implantée par les auteurs :

- Les auteurs affirment que leurs tests suggèrent une *faible* perte de performance de l'ordre de 6 à 20%. En effet, une perte de performance de 6% peut être acceptable dans la plupart des cas mais une perte de 20% est assez élevée à notre avis. Les auteurs devraient voir à optimiser leur approche si possible.

- *Plus important que les potentiels problèmes de performance, l'approche en soi semble assez sensible*. En effet, elle nécessite la modification des classes du système de la plate-forme Java, c'est-à-dire des classes de la librairie standard de Java. Ces modifications sont parfois assez complexes et peuvent être effectuées directement dans le code objet exécutable des fichiers de classe et non seulement dans le regroupement de constantes. Les classes du système sont cependant au centre même de la plate-forme. Le bon fonctionnement de celle-ci dépend beaucoup du bon fonctionnement des classes du système. Ainsi, quoiqu'il est certainement possible de faire des modifications qui ne créeront pas de bogues, une telle approche dans le domaine de la sécurité informatique est assez difficile à accepter d'emblée pour les risques qu'elle entraîne (pensons, par exemple, aux interactions possibles avec l'initialisation des objets et la sûreté des sous-routines). Le simple fait que l'administrateur ait à implanter les tests à effectuer pour contrôler l'exécution des programmes est une philosophie sensible aux erreurs.

  De plus, cet administrateur doit théoriquement modifier toutes les classes critiques du système, c'est-à-dire toutes celles qui accèdent directement aux ressources qu'il veut protéger. Pour une plate-forme telle que Java, cette tâche est ardue et un simple oubli ouvre une brèche dans la sécurité.

  De surcroît, il peut être difficile de fixer une limite supérieure à l'utilisation de ressources telles que la mémoire ou les fenêtres. En effet, un programme tout à fait légitime pourrait nécessiter l'ouverture de plusieurs fenêtres et les classes du système peuvent elles-mêmes avoir besoin de beaucoup de mémoire.

  Finalement, le paradigme fondamental utilisé par cette approche est le même que pour les antivirus. En effet, dans le monde des antivirus, lorsqu'un nouveau virus

est découvert, sa signature est calculée et ajoutée à une base de données. La base de données locale de chaque utilisateur doit ensuite être mise à jour de façon régulière. Dans le cas de cette approche, lorsqu'une nouvelle attaque est découverte, l'administrateur doit modifier ses classes afin d'en tenir compte. Ce paradigme est de moins en moins accepté, car on se rend vite compte que les nouveaux virus et attaques peuvent faire beaucoup de dégât en très peu de temps, moins qu'il n'en faut pour réagir et mettre notre sécurité à jour.

4.4 Conclusion du deuxième article

En terminant, il est vrai que cette approche apporte beaucoup de flexibilité à la sécurité de la plate-forme Java mais au prix des risques qu'elle comporte. Ce n'est donc pas une approche qu'on adopterait ni même une approche pour laquelle on opterait pour faire de la recherche.

Cet article montre cependant qu'il y a un besoin réel dans la définition de politiques de sécurité flexibles et expressives. D'où la pertinence d'étudier l'article présenté dans la section suivante.

5 Troisième article – Supporting Reconfigurable Security Policies for Mobile Programs

Ce troisième et dernier article présente une infrastructure de sécurité bâtie autour d'un mécanisme « événement/réponse ». Cette infrastructure possède un langage de spécification des politiques de sécurité expressif et assez intuitif. Ces politiques de sécurité sont renforcées en instrumentant le code objet Java des programmes à surveiller. L'avantage majeur de cette infrastructure selon les auteurs est que les politiques de sécurité peuvent être changées et reconfigurées de façon dynamique et même automatique durant l'exécution des programmes à vérifier. Encore une fois, les politiques définies à l'aide de cette infrastructure sont plus puissantes que celles pouvant être définies actuellement dans la plate-forme Java.

5.1 Infrastructure

L'infrastructure de sécurité présentée dans cet article possède un avantage majeur : les politiques de sécurité qu'on y définit sont dynamiques, c'est-à-dire qu'elles peuvent changer durant l'exécution des programmes surveillés. Avec des politiques de sécurité statiques, l'administrateur doit en quelque sorte prévoir tous les coups possibles ou du moins, il doit s'assurer de maintenir ses politiques à jour. Ceci est plus facile à gérer avec des politiques dynamiques.

Mais avant d'étudier plus en profondeur les détails de cette infrastructure, il est de mise d'effectuer un bref survol de cette infrastructure et de définir certains termes.

5.1.1 Principe de base

En général, un programme accède à une ressource en appelant les méthodes appropriées du système. Par exemple, un programme *P* accède à une ressource *R* en appelant la méthode *m*. Très souvent, aucune restriction n'empêche le programme d'appeler la méthode. L'infrastructure présentée dans cet article permet justement de rendre l'appel de la méthode conditionnel. Ainsi, un programme *P* ne pourra pas toujours appeler la méthode *m* pour accéder à la ressource *R*.

Un point positif de cette infrastructure est que ces contraintes sont définies indépendamment de *P* et *m*. Elles sont renforcées en injectant du nouveau code dans *P* et *R* avant que ces entités soient chargées en mémoire par la machine virtuelle. De plus, les contraintes peuvent être modifiées dynamiquement et le code injecté sera modifié en conséquence. Les entités concernées seront également rechargées afin de prendre les modifications en considération. Cette capacité est très appréciée dans des contextes où les politiques de sécurité doivent évoluer en fonction des conditions d'opération ou des objectifs organisationnels.

5.1.2 Définition d'une ressource

Dans le cadre de cette infrastructure, une *ressource* peut représenter n'importe quel module logiciel. Par exemple, une ressource peut être une méthode, une classe ou un ensemble de classes protégé par une politique de sécurité. Les ressources conceptuelles ou physiques, telles que des bases de données ou des imprimantes, doivent être encapsulées par une méthode ou une classe Java pour être protégées par une politique de cette infrastructure.

5.1.3 Définition d'un acteur (principal)

Toujours dans le cadre de cette infrastructure, un *acteur* (aussi appelé *principal*) représente aussi une méthode, une classe ou un groupe de classes. Par exemple, *www.sun.com* représente un acteur qui comprend toutes les classes chargées à partir de cette origine. Les acteurs sont donc définis en énumérant les classes qu'ils comprennent ou en définissant une méthode permettant de déterminer si une classe donnée est comprise dans un acteur ou pas. Cette spécification des acteurs est très importante, car l'administrateur doit s'assurer qu'elle ne permet pas à un intrus de répondre aux critères d'un certain acteur et de se faire passer pour cet acteur.

5.1.4 Définition d'un événement

Un *événement* survient à chaque fois qu'un acteur *A* (équivalent au programme *P* mentionné auparavant) accède à une ressource *R*. Un événement peut contenir une condition, auquel cas cette condition devra être vraie pour que l'événement survienne.

5.1.5 Définition d'une réponse

Une *réponse* décrit les actions à effectuer avant et/ou après qu'un événement particulier survienne. Par exemple, une exception peut être levée ou un journal peut être maintenu.

5.1.6 Mécanisme « événement/réponse »

Le mécanisme « événement/réponse » régit les politiques de sécurité de cette infrastructure. En effet, une politique de sécurité est spécifiée par trois informations : les événements que l'infrastructure doit surveiller, les conditions sous lesquelles ces événements nécessitent une réponse et les réponses qui doivent être effectuées. Ainsi, les acteurs et les ressources sont liés par des événements qui engendrent des réponses selon l'évaluation de certaines conditions.

Étudions maintenant le langage de spécification des politiques de sécurité de cette infrastructure de sécurité.

5.1.7 Langage de spécification des politiques de sécurité

Ce qui suit constitue la grammaire du langage de spécification des politiques de sécurité de l'infrastructure étudiée :

```
Policy            ::= { PolicyStatements | Definitions }
PolicyStatements  ::= { Constraint | AddStatement | EnableStatement }
Definitions       ::= { PolicyGroup | GroupStatement }
Constraint        ::= before Event do Response
                    | after Event do Response
AddStatement      ::= add Type Name to ClassName
EnableStatement   ::= enable Event
Response          ::= ResponseName '(' ParameterList ')'
Event             ::= [Entity] Invocation Entity [and Condition]
Invocation        ::= -->
Entity            ::= class ClassName
                    | method MethodName '(' ParameterList ')'
                    | group GroupName
Condition         ::= BooleanExpression
PolicyGroup       ::= policy PolicyName
                      '{' PolicyStatements ';' { PolicyStatements } '}'
GroupStatement    ::= group GroupName '{' Entity ';' { Entity } '}'
                    | define group GroupName
                      '{' FunctionName '(' Parameters ')' '}'
```

Ce langage définit essentiellement des relations événement/réponse et permet de grouper des entités pour faciliter la spécification des politiques. Voyons un exemple. Supposons que la classe suivante a été définie :

```
class GroupFunction
{
  static boolean RogueSite(Class ncl)
  {
    URL url = ncl.getClassLoader().getURL();
    String name = url.toString();
    return name.equals("http://www.roguesite.com");
  }
}
```

Cette classe déclare une méthode permettant de déterminer si une classe donnée est un acteur `RogueSite`.

Supposons maintenant qu'un administrateur veut écrire dans un journal tous les accès fichier effectués par des programmes provenant de `www.roguesite.com`. Une politique permettant d'obtenir ce comportement pourrait se définir comme suit :

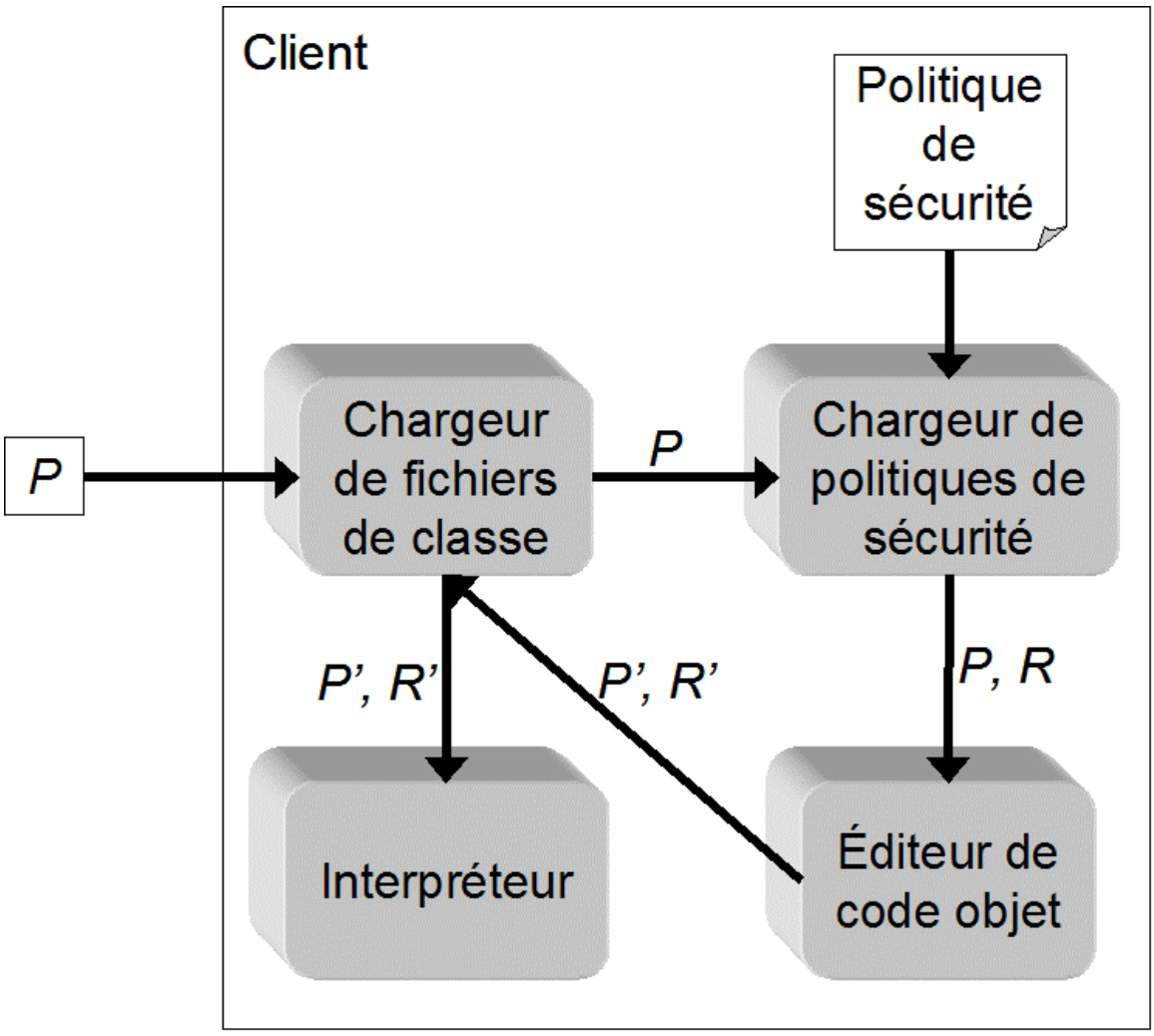
```
group ReadFile
{
  class FileInputStream;
  class FileOutputStream;
  …
}

group RogueSite
{
  GroupFunction.RogueSite(Class newClass);
}

after group RogueSite --> group ReadFile do AuditResponse()
```

Ainsi, chaque fois qu'une des classes de fichier listées dans l'acteur `ReadFile` sera utilisée par une classe de l'acteur `RogueSite`, un journal sera maintenu en effectuant la réponse `AuditResponse`.

Voyons maintenant comment ces politiques de sécurité sont renforcées par l'infrastructure.

5.1.8 Renforcement des politiques de sécurité

L'infrastructure de sécurité renforce les politiques de sécurité en injectant du nouveau code objet entre l'acteur accédant à une ressource et la ressource elle-même. Ce nouveau code vérifie si l'accès est accepté. Il est ajouté avant que les classes modifiées soient chargées en mémoire par la machine virtuelle Java. L'architecture suivante illustre le fonctionnement de l'infrastructure :



Lorsqu'un programme mobile *P* est téléchargé sur l'ordinateur du client, il est pris en charge par le chargeur de fichiers de classe de la machine virtuelle. Cependant, ce chargeur a été modifié par les auteurs de façon à ce qu'il transmette les fichiers chargés au chargeur de politiques de sécurité, qui, selon la politique de sécurité en cours, transmet à son tour le programme *P* augmenté de la (des) ressource(s) *R* accédée(s) à l'éditeur de code objet. Ce dernier injecte le code permettant de vérifier si les accès à la (aux) ressource(s) *R* sont permis. Les versions modifiées de *P* et *R* sont *P'* et *R'*. Ces versions sont retransmises au chargeur de fichiers de classe qui les transmet à l'interpréteur qui les exécute. Si le chargeur de fichiers de classe avait déjà chargé une version de *P* et de *R*, *P'* et *R'* les remplacent.

La nature du code injecté dépend de la politique de sécurité en cours. Par exemple, supposons que la politique est constituée d'une paire événement/réponse nommée `evenement` et `reponse`. De plus, l'événement est défini par un triplet `<sujet, cible, condition>`. Cette politique serait implantée par l'injection du code suivant :

```
if (caller == evenement.sujet &&
    evenement.condition)
{
  reponse.doResponse();
}
```

5.2 Analyse

Les auteurs du présent article ont analysé leur infrastructure de façon à détecter des brèches de sécurité potentielles dans leur approche. Trois possibilités ont été étudiées :

1. Grâce aux changements effectués par les auteurs au niveau du chargeur de fichiers de classe de la machine virtuelle, une classe peut être modifiée dynamiquement. Cette capacité était nécessaire pour pouvoir modifier les politiques de sécurité dynamiquement et effectuer les transformations nécessaires au niveau du code injecté. Cependant, cette capacité pourrait aussi éventuellement être utilisée par une applet malicieuse afin de contourner une politique établie. Par exemple, cette applet malicieuse pourrait définir une nouvelle méthode `newWrite` dans le cas où la méthode `write` usuelle est protégée. Les auteurs ont solutionné ce problème en séparant les différentes composantes des programmes Java dans des espaces distincts (de l'anglais « namespaces ») les uns des autres. Par exemple, les classes du système, les ressources et les classes de l'utilisateur se retrouvent dans des espaces distincts. De plus, le chargeur de fichiers de classe ne permet pas à une classe donnée de modifier une autre classe qui ne se retrouve pas dans le même espace. Ainsi, les classes de l'utilisateur ne peuvent pas modifier les ressources ou les classes du système.

   La sécurité du système pourrait aussi être compromise en utilisant le chargeur de politiques de sécurité pour modifier dynamiquement la politique en cours. Les auteurs ont corrigé ce problème en instaurant la même contrainte au niveau du chargeur de politiques de sécurité que celle établie au niveau du chargeur de fichiers de classe, c'est-à-dire qu'une classe de l'utilisateur ne peut pas modifier une politique définie au niveau du système par un chargeur de politiques de sécurité du système.

   Cependant, une applet peut encore modifier une politique de sécurité du système en accédant au chargeur de politiques de sécurité de façon indirecte, par une ressource comme le système de fichiers, par exemple. Pour solutionner ce problème, les auteurs proposent aux administrateurs de protéger toutes les ressources à l'aide d'une politique de sécurité qui empêche ce genre d'attaque.

2. Certaines infrastructures de sécurité sont vulnérables à des attaques utilisant la réflexion, c'est-à-dire la possibilité qu'offre Java de découvrir les informations contenues dans les classes chargées en mémoire. Par exemple, un programme Java peut découvrir dynamiquement les noms des méthodes des classes chargées en mémoire à un instant donné. En utilisant cette capacité, une applet malicieuse

peut connaître les noms des méthodes d'une classe encapsulée dans une autre qui la protège. L'infrastructure décrite ici n'est pas vulnérable à ce genre d'attaques, car elle n'encapsule pas les classes critiques dans d'autres classes protectrices mais modifie plutôt directement ces classes critiques.

3. Finalement, sous certaines infrastructures, il est possible d'exploiter la synchronisation des processus légers (de l'anglais « threads ») pour contourner la sécurité. Par exemple, si un compteur est utilisé pour cumuler le nombre de fenêtres ouvertes à un instant donné et que ce compteur n'est pas synchronisé, un processus léger malicieux pourrait tenter de créer plus de fenêtres que la limite permise en exploitant le fait que la lecture et l'écriture dans le compteur ne se fait pas de façon atomique. Encore une fois, pour éviter ce problème, les auteurs ont fait en sorte que tout le code injecté dans les classes soit synchronisé.

Les auteurs ont aussi analysé la performance de leur infrastructure, résumée ci-après.

5.3 Performance

Les auteurs ont d'abord évalué le temps moyen pris pour ajouter et retirer dynamiquement une politique de sécurité. Ils ont aussi calculé le temps moyen pris pour charger une classe originale en mémoire et pour charger la même classe mais modifiée. Le temps moyen pris pour appeler une méthode originale et pour appeler la même méthode mais modifiée a aussi été estimé. Ces données sont compilées dans le tableau suivant :

Ajouter une politique	0,060 s
Retirer une politique	0,015 s
Charger une classe originale en mémoire	0,018 s
Charger la même classe modifiée en mémoire	0,051 s
Appeler une méthode originale	0,000070 s
Appeler la même méthode modifiée	0,00092 s

Les performances à ce niveau sont donc inacceptables. Les auteurs n'en font pas mention mais dans le cas du chargement d'une classe, une augmentation du temps d'exécution de 183% due aux modifications est observée. Dans le cas de l'appel d'une méthode, c'est une augmentation de 1 214% qui est observée !

Les auteurs ont également fait des tests sur différentes applications incluses dans le jeu de tests SPECjvm '98. Ils ont calculé le temps pris pour exécuter ces applications sans politique de sécurité (T_o) et avec une politique de sécurité (T_p). Ils ont cependant fait une erreur dans le calcul de l'augmentation du temps d'exécution. En effet, au lieu de calculer (T_p - T_o) / T_o, ils ont calculé (T_p - T_o) / T_p. Évidemment, cela sous-estime les augmentations du temps d'exécution et fausse certains commentaires qu'ils donnent à ce niveau. Le tableau qui suit est celui qu'on aurait dû retrouver dans l'article :

Applications	Temps moyen d'exécution avec MVJ originale (s)	Temps moyen d'exécution avec MVJ modifiée pour supporter les politiques de sécurité (s)	$(T_p - T_o) / T_o$
check	1,06	1,497	0,412
mtrt	1709,399	1845,753	0,0798
jess	1420,888	1532,356	0,0784
compress	7966,578	8219,17	0,0317
db	2675,772	2963,061	0,107
mpegaudio	6383,705	6746,463	0,0568
jack	2083,441	2804,971	0,346
javac	1682,285	1820,105	0,0819

Dans les cas où il n'y a pas beaucoup de différence entre T_o et T_p, l'erreur de calcul n'a pas de répercussions trop importantes. Cependant, pour *check* et *jack*, les augmentations du temps d'exécution sont respectivement de 41% et 35%, et non pas 29% et 26%, comme énoncé dans l'article. Ces augmentations du temps d'exécution sont donc plus importantes que les auteurs le prétendent et des efforts d'optimisation devraient être envisagés.

5.4 Limites actuelles

À la lecture de cet article, il a été possible de mettre en lumière quelques limites à cette infrastructure de sécurité utilisant un mécanisme « événement/réponse », telle qu'implantée par les auteurs :

- *La performance de l'infrastructure est certainement un point à améliorer*. Le fait que tout le code injecté soit synchronisé ne doit pas aider, car ainsi, les gains de performance obtenus par pseudo-parallélisme sont perdus. De plus, la lourdeur de leur approche engendrée par l'hyper-dynamisme des politiques de sécurité est certainement responsable d'une grande partie des augmentations du temps d'exécution.

- *Tout comme pour l'article précédent, l'approche en soi semble sensible*. En effet, elle nécessite aussi la modification des classes du système de la plate-forme Java, c'est-à-dire des classes de la librairie standard de Java (voir sous-section 4.3).

  De plus, l'administrateur doit théoriquement lister toutes les classes critiques de la librairie standard de Java dans ses politiques pour assurer une pleine sécurité des ressources, ce qui n'est pas facile en pratique. Ce problème est omniprésent dans la sécurité de Java et pose des difficultés à plusieurs infrastructures de sécurité. Dans le cas de l'infrastructure en question, les auteurs mentionnent que toutes les ressources doivent être correctement protégées pour assurer que la sécurité ne pourra pas être contournée en accédant au chargeur de politiques par le biais du

système de fichiers, par exemple. Ce niveau d'assurance semble assez difficile à atteindre en pratique.

Finalement, beaucoup de changements semblent avoir été effectués au niveau de la machine virtuelle Java pour supporter les politiques de sécurité dynamiques. De plus, la définition de ces politiques demande des efforts de programmation assez importants. Ainsi, il semble que cette infrastructure soit sujette à souffrir de plusieurs problèmes reliés aux erreurs d'implantation. Dans le domaine de la sécurité informatique, ce n'est pas une propriété souhaitable.

5.5 Conclusion du troisième article

En conclusion, l'infrastructure de sécurité présentée dans cet article tente d'intégrer des politiques de sécurité plus flexibles, expressives et dynamiques dans la plate-forme Java. Cependant, la performance ne semble pas assez grande pour que l'approche soit utilisable en pratique. De plus, la complexité de cette approche fait en sorte qu'il est difficile de lui faire confiance dans des contextes critiques. Dans le domaine de la sécurité informatique, les solutions les plus simples sont souvent les meilleures.

6 Conclusion

Les trois articles présentés dans ce document montrent que la sécurité de la plate-forme Java est particulièrement bien conçue, car les nouvelles approches exposées n'ont pas été très concluantes. En fait, l'architecture de la sécurité de Java ne demande qu'à être utilisée adéquatement. Le niveau de confiance qu'elle apporte est généralement assez élevé pour assurer la sécurité d'un environnement courant. Cependant, pour les systèmes critiques et/ou particulièrement exposés, l'architecture de la sécurité de Java doit être étendue pour offrir une plus grande assurance. Toutefois, ces extensions ne doivent pas sacrifier la robustesse. Il reste donc encore de la place pour la recherche dans le domaine de la sécurité de Java.

Les principaux enjeux à surveiller durant les prochaines années dans ce domaine de recherche sont :

1. la fusion des analyses statiques et dynamiques afin de jouir des avantages des deux approches,

2. la formalisation des systèmes de sécurité pour atteindre une plus grande robustesse et une meilleure confiance,

3. l'évaluation du risque résiduel pour mieux connaître les faiblesses des systèmes de sécurité et

4. les politiques de sécurité formelles, intuitives et uniformes pour définir avec précision les comportements acceptables et inacceptables des logiciels.

De plus, la modélisation certifiée, qui a pour but d'intégrer la sécurité le plus tôt possible dans le cycle de développement, et les certificats de conformité, qui donnent l'assurance de la correction d'une implantation par rapport à sa spécification, sont certainement des sujets d'avenir.

Finalement, il va sans dire que la collaboration internationale entre chercheurs est un atout non négligeable, voire même nécessaire, pour valider le développement dans ce domaine de recherche assez complexe.

Bibliographie

[Aho et al.86] Aho Alfred V., Sethi Ravi & Ullman Jeffrey D., 1986 : *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, USA, janvier 1986, 796 pages.

[Basin et al.02] Basin David, Friedrich Stefan, Gawkowski Marek & Posegga Joachim, 2002 : « Bytecode Model Checking: An Experimental Analysis ». Dans les actes de *Model Checking Software, 9th International SPIN Workshop*, Grenoble, France, avril 2002, pp. 42-59.

[Chander et al.01] Chander Ajay, Mitchell John C. & Shin Insik, 2001 : « Mobile Code Security by Java Bytecode Instrumentation ». Dans les actes de *DARPA Information Survivability Conference and Exposition II (DISCEX'01)*, Anaheim, CA, USA, juin 2001, vol. 2, pp. 27-40.

[Hashii et al.00a] Hashii Brant, Malabarba Scott, Pandey Raju & Bishop Matt, 2000 : « Supporting Reconfigurable Security Policies for Mobile Programs ». Dans les actes de *Ninth International World Wide Web Conference*, Amsterdam, Pays-Bas, mai 2000, 22 pages.

[Hashii et al.00b] Hashii Brant, Malabarba Scott, Pandey Raju & Bishop Matt, 2000 : « Supporting Reconfigurable Security Policies for Mobile Programs ». *Computer Networks*, Pays-Bas, juin 2000, vol. 33, no. 1-6, pp. 77-93.

[Holzmann97] Holzmann Gerard J., 1997 : « The SPIN Model Checker ». *IEEE Transactions on Software Engineering*, mai 1997, vol. 23, no. 5, pp. 279-295.

[Lindholm & Yellin99] Lindholm Tim & Yellin Frank, 1999 : *The Java Virtual Machine Specification*. The Java Series, Addison-Wesley, Reading, MA, USA, avril 1999, 473 pages.

[Malabarba et al.00] Malabarba Scott, Pandey Raju, Gragg Jeff, Barr Earl & Barnes J. Fritz, 2000 : « Runtime Support for Type-Safe Dynamic Java Classes ». Dans les actes de *European Conference on Object-Oriented Programming*, Sophia Antipolis et Cannes, juin 2000.

[McMillan92] McMillan Ken, 1992 : *Symbolic Model Checking: An Approach to the State Explosion Problem*. Thèse de doctorat, Département d'informatique, Carnegie Mellon University, Pittsburgh, PA, USA, mai 1992.

[Pandey & Hashii99] Pandey Raju & Hashii Brant, 1999 : « Providing Fine-Grained Access Control for Java Programs ». Dans les actes de *13th Conference on Onject-Oriented Programming*, Lisbone, juin 1999.

[Shin & Mitchell01] Shin Insik & Mitchell John C., 2001 : « Java Bytecode Modification and Applet Security ». *Stanford CS Tech Report*, 20 pages, http://theory.stanford.edu/people/jcm/publications.htm.

[Wahbe et al.93] Wahbe Robert, Lucco Steven, Anderson Thomas E. & Graham Susan L., 1993 : « Efficient Software-Based Fault Isolation ». Dans les actes de *14th Symposium on Operating Systems Principles*, décembre 1993.

[Yellin95] Yellin Frank, 1995 : « Low Level Security in Java ». Dans les actes de *World Wide Web Journal: The Fourth International World Wide Web Conference*, Cambridge, MA, USA, 1995, pp. 369-380.

Liste de distribution

DISTRIBUTION INTERNE
RDDC – Valcartier TN 2002 - 154

1 - Directeur général
1 - Directeur général adjoint
3 - Bibliothèque des documents
1 - Chef/Gestion de l'Information et de la Connaissance
1 - Chef/Systèmes de Systèmes
1 - F. Painchaud (auteur)
1 - D. Demers
1 - R. Charpentier
1 - M. Salois
1 - G. Turcotte
1 - M. Lizotte
1 - F. Lemieux
1 - M. Gauvin

DISTRIBUTION EXTERNE
RDDC – Valcartier TN 2002 - 154

1 - Directorate R & D – Knowledge and Information Management
1 - Defence R & D Canada – Headquarters
4 - Director General Joint Force Development
1 - Director Sciences and Technology – Command and Control Information Systems
2 - Directorate Sciences and Technology – Command and Control Information Systems
2 - Directorate Distributed Computing Engineering and Integration
1 - Directorate Land Command Systems Program Management
2 - Directorate Land Requirements
1 - Assistant Deputy Minister (Information Management)
1 - Canadian Forces Experimentation Center
1 - Canadian Forces Command and Staff College
Toronto
Attn : Commanding Officer
1 - Canadian Forces Maritime Warfare School
Canadian Forces Base Halifax
Halifax, Nova Scotia
Attn : Commanding Officer
1 - Communications Security Establishment
Attn : Louis Bélanger
1 - Communications Security Establishment
Attn : Sylvain Bazinet

DISTRIBUTION EXTERNE (SUITE)
RDDC – Valcartier TN 2002 - 154

5 - Defence R & D Canada –
Ottawa
Attn : Mark McIntyre

FICHE DE CONTRÔLE DU DOCUMENT

1. PROVENANCE (le nom et l'adresse)	2. COTE DE SÉCURITÉ (y compris les notices d'avertissement, s'il y a lieu)
Frédéric Painchaud R & D pour la défense Canada - Valcartier 2459, boul. Pie-XI Nord Val-Bélair, QC G3J 1X5	SANS CLASSIFICATION

3. TITRE (Indiquer la cote de sécurité au moyen de l'abréviation (S, C, R ou U) mise entre parenthèses, immédiatement après le titre.)
Nouvelles approches en sécurité de Java (U)

4. AUTEURS (Nom de famille, prénom et initiales. Indiquer les grades militaires, ex.: Bleau, Maj. Louis E.)
Painchaud, Frédéric

5. DATE DE PUBLICATION DU DOCUMENT (mois et année)	6a. NOMBRE DE PAGES	6b. NOMBRE DE REFERENCES
10/2002	52	13

7. DESCRIPTION DU DOCUMENT (La catégorie du document, par exemple rapport, note technique ou mémorandum. Indiquer les dates lorsque le rapport couvre une période définie.)
Note technique

8. PARRAIN (le nom et l'adresse)

9a. NUMÉRO DU PROJET OU DE LA SUBVENTION (Spécifier si c'est un projet ou une subvention)	9b. NUMÉRO DE CONTRAT
15BF34	

10a. NUMÉRO DU DOCUMENT DE L'ORGANISME EXPÉDITEUR	10b. AUTRES NUMÉROS DU DOCUMENT
RDDC - Valcartier TN 2002 - 154	N/A

11. ACCÈS AU DOCUMENT (Toutes les restrictions concernant une diffusion plus ample du document, autres que celles inhérentes à la cote de sécurité.)

- ☒ Diffusion illimitée
- ☐ Diffusion limitée aux entrepreneurs des pays suivants (spécifier)
- ☐ Diffusion limitée aux entrepreneurs canadiens (avec une justification)
- ☐ Diffusion limitée aux organismes gouvernementaux (avec une justification)
- ☐ Diffusion limitée aux ministères de la Défense
- ☐ Autres

12. ANNONCE DU DOCUMENT (Toutes les restrictions à l'annonce bibliographique de ce document. Cela correspond, en principe, aux données d'accès au document (11). Lorsqu'une diffusion supplémentaire (à d'autres organismes que ceux précisés à la case 11) est possible, on pourra élargir le cercle de diffusion de l'annonce.)

13. SOMMAIRE (Un résumé clair et concis du document. Les renseignements peuvent aussi figurer ailleurs dans le document. Il est souhaitable que le sommaire des documents classifiés soit non classifié. Il faut inscrire au commencement de chaque paragraphe du sommaire la cote de sécurité applicable aux renseignements qui s'y trouvent, à moins que le document lui-même soit non classifié. Se servir des lettres suivantes: (S), (C), (R) ou (U). Il n'est pas nécessaire de fournir ici des sommaires dans les deux langues officielles à moins que le document soit bilingue.)

Dans ce document, trois articles portant sur différents aspects de la sécurité du langage de programmation Java sont résumés. Ces articles ont été récemment publiés dans des actes de conférences et des journaux internationaux. Le premier article présente une approche visant à remplacer l'analyse de flot de données effectuée par le vérificateur de code objet de la machine virtuelle Java. Cette approche est basée sur le « model checking ». Le deuxième article présente une approche visant à instrumenter le code objet Java afin d'assurer une meilleure sécurité des applets et des services Jini. Cette nouvelle approche permet de détecter et d'arrêter certains types de dénis de service, d'assurer l'intégrité des données critiques et la confidentialité du système et même de prévenir certains types de personnification. Finalement, le troisième et dernier article présente une infrastructure de sécurité utilisant un mécanisme « événement/réponse ». Cette infrastructure possède un langage de spécification des politiques de sécurité expressif et plutôt intuitif.

Puisque chaque article présente une nouvelle approche pour résoudre un des problèmes de l'architecture de sécurité de Java, il semble essentiel de les résumer afin de souligner leur(s) application(s). Pour terminer, quelques-unes des limites, surmontables ou non, de ces nouvelles approches sont données et quelques commentaires personnels sont transmis.

14. MOTS-CLÉS, DESCRIPTEURS OU RENSEIGNEMENTS SPÉCIAUX (Expressions ou mots significatifs du point de vue technique, qui caractérisent un document et peuvent aider à le cataloguer. Il faut choisir des termes qui n'exigent pas de cote de sécurité. Des renseignements tels que le modèle de l'équipement, la marque de fabrique, le nom de code du projet militaire, la situation géographique, peuvent servir de mots-clés. Si possible, on doit choisir des mots-clés d'un thésaurus, par exemple le "Thesaurus of Engineering and Scientific Terms (TESTS)". Nommer ce thésaurus. Si l'on ne peut pas trouver de termes non classifiés, il faut indiquer la classification de chaque terme comme on le fait avec le titre.)

Information Technology Security, Software Security, Program Verification, Program Instrumentation, Mobile Code, Java, Java Security, Java Security Architecture, Security Infrastructure, Java Bytecode Verifier.