RenPar'19 / SympA'13 / CFSE'7
Toulouse, France, du 9 au 11 septembre 2009

I-JVM : une machine virtuelle Java pour l'isolation de composants dans OSGi

Nicolas Geoffray*, Gaël Thomas, Gilles Muller, Pierre Parrend, Stéphane Frénot, Bertil Folliot

*Laboratoire d'Informatique de Paris 6
104 avenue du Président Kennedy
75016 PARIS - France
nicolas.geoffray@lip6.fr

Résumé
OSGi est une plateforme orientée composants implémentée en Java qui est de plus en plus utilisée pour le développement d'applications extensibles. Cependant, les machines virtuelles Java existantes ne sont pas capables d'isoler des composants entre eux. Par exemple, un composant malicieux peut bloquer l'exécution de la plateforme en allouant trop de mémoire ou modifier le comportement d'autres composants en modifiant des variables globales. Nous présentons I-JVM, une machine virtuelle Java qui offre une isolation légère entre composants tout en préservant la compatibilité avec les applications OSGi existantes. I-JVM résoud les 8 vulnérabilités connues sur la plateforme OSGi liées à la machine virtuelle, et ne diminue que de 20% les performances des applications en comparaison avec la machine virtuelle sur laquelle elle est implémentée.

Mots-clés : Java, Isolation, Migration de threads, Comptage de ressources, OSGi

1. Introduction

OSGi [26] est une plateforme Java orientée composants qui est de plus en plus utilisée comme environnement d'exécution pour applications extensibles comme Eclipse [2] ou Jonas [3]. L'extensiblité d'OSGi tient à son architecture composée de *bundles*, un bundle étant un ensemble de composants chargés par un class loader Java spécifique. La plateforme OSGi est populaire parce qu'elle offre modularité et performance des communications entre bundles avec l'appel de méthode direct.
Au départ, OSGi a été développé pour des environnements où les bundles se font confiance. Aujourd'hui, OSGi est aussi utilisé comme une plateforme pour les set-top-box où des bundles développés par des parties tierces sont téléchargés et installés à la demande [31]. OSGi ne protège cependant pas ou peu les bundles et la plateforme d'un bundle mal écrit ou malicieux. Premièrement parce que les objects *java.lang.Class*, les chaines de caractères Java (*java.lang.String*) et les variables *static* sont partagées dans une machine virtuelle Java (JVM). Modifier une de ces variables impacte l'exécution de tous les bundles. Deuxièmement, un bundle peut monopoliser le CPU ou prendre trop de mémoire et ainsi faire du déni de service. Troisièmement, il peut être impossible de tuer un bundle qui fait du déni de service, ce qui fait qu'arrêter la plateforme est la seule solution envisageable.
Un travail récent de Pierre Parrend et Stéphane Frénot [29] a identifé 25 vulnérabilités dans les implémentations actuelles de la plateforme JVM/OSGi. Ces vulnérabilités sont soit de l'ordre de la corruption de données ou du déni de service. 17 de ces vulnérabilités sont dûes à l'implémentation d'OSGi luimême et peuvent être résolues en ajoutant des tests de sécurité. Cependant, les 8 autres vulnérabilités concernent l'isolation des bundles et doivent être résolues au niveau de la JVM.
Plusieurs approches existent pour ajouter de l'isolation dans les JVM, à travers des *Isolates* (ou processus Java) [5, 9, 17]. Ces approches dupliquent les objects *java.lang.Class*, *java.lang.String* et les variables *static* entre les isolates. Cependant, pour assurer une isolation complète, ces solutions limitent l'exécution d'un thread dans un seul isolate. Par conséquent, les communications entre isolates nécessitent un mécanisme RPC, avec passage de paramètres par copie et synchronisation entre threads. Utiliser des

RPCs dans une plateforme OSGi où les appels entre bundles sont omni-présents, provoquerait une perte de performances conséquente. Par exemple, malgré une machine virtuelle optimisée, les développeurs de Singularity [12] indiquent qu'un simple RPC à travers un tas partagé se fait en 2500 cycles, alors qu'un appel direct se fait en 10 cycles. De plus, le passage de paramètres par copie demande à modifier les bundles existants au niveau source ou bytecode [34], ce qui pourrait compromettre la compatibilité entre les bundles.

Dans cet article, nous présentons I-JVM, une JVM avec la notion d'isolates *légers* conçus pour supporter les besoins d'OSGi en associant à chaque bundle un isolate. La contribution d'I-JVM est de permettre la migration de threads entre isolates, de manière à maintenir le faible coût d'une communication entre bundles. I-JVM n'impose pas de modification des communications inter-bundles et est compatible avec les bundles existants. Les avantages de I-JVM sont :

– Isolation mémoire : le projet Multi-Tasking Virtual Machine (MVM) [9] a montré que dupliquer les objects *java.lang.Class*, *java.lang.String* et les variables *static* était suffisant pour isoler la mémoire entre isolates. Par conséquent, un isolate ne peut pas accéder aux objets d'un autre isolate à moins que ce dernier ne lui ait donné des objets en paramètres d'une méthode.
– Comptage de ressources : I-JVM maintient pour chaque thread l'isolate dans lequel il s'exécute, ce qui permet de compter les allocations mémoires et l'utilisation CPU par isolate. Ces informations peuvent être utilisées par un administrateur ou par des politiques de ressources pour détecter des attaques de déni de service.
– Terminaison d'isolates : lorsqu'un isolate se termine, les méthodes qu'il définit ne doivent plus être appelées ni exécutées. Dans le cas où un thread retourne d'une méthode vers une méthode d'un isolate qui se termine, I-JVM modifie la pile d'exécution pour lever une exception qui ne pourra être attrapée que par un isolate qui ne se termine pas. Tous les objets référencés par l'isolate sont détruits, exceptés les objets partagés avec les autres isolates.

I-JVM a été implémenté avec la JVM du projet VMKit [15, 16], une JVM écrite pour faciliter des expérimentations du langage ou de la plateforme Java. Nous avons utilisé I-JVM pour exécuter deux plateformes OSGi existantes : Felix [1] de la fondation Apache et Equinox [2] du projet Eclipse.

Les résultats de notre travail sont :

– I-JVM résoud les vulnérabilités JVM identifées dans [29]. Nous présentons 8 attaques qui couvrent ces vulnérabilités.
– I-JVM a un coût à l'exécution de 16% pour les appels inter-isolates. Ce résultat est largement inférieur au coût d'un RPC entre deux processus. I-JVM diminue les performances de 1% à 20% par rapport à VMKit sur une suite de macro-benchmarks.
– I-JVM ajoute 650 lignes de code à VMKit. Implémenter les concepts d'I-JVM dans une autre JVM devrait demander autant d'ajout de code.

La suite de l'article est composée comme suit. La section 2 décrit les vulnérabilités de la plateforme JVM/OSGi. La section 3 décrit les concepts et l'implémentation d'I-JVM. La section 4 évalue les performances d'I-JVM ainsi que sa robustesse contre les attaques de déni de service. La section 5 décrit les travaux connexes et la section 6 conclue l'article.

2. Vulnérabilités et attaques dans OSGi

Parrend *et al.* ont identifié trois types de vulnérabilités [29] : (i) au niveau du système d'exploitation, (ii) au niveau de la plateforme OSGi et (iii) au niveau de la JVM. Le premier type de vulnérabilités est dû à la possibilité d'exécuter du code natif non vérifié dans la JVM ou dans un autre processus. Ces vulnérabilités sont provoquées par l'interface de programmation JNI ou la méthode *Runtime.exec*. Le second type de vulnérabilités est dû à des erreurs d'implémentation dans le runtime OSGi. Ces vulnérabilités peuvent être résolues en ajoutant des tests de sécurité [29]. Dans cet article, nous visons à résoudre le troisième type de vulnérabilités, qui est au niveau de la JVM.

Les vunérabilités JVM peuvent elles-même être divisées en trois catégories : (i) Manque d'isolation, (ii) Manque de gestion de ressources et (iii) Impossibilité de terminer l'exécution d'un bundle. Dans cette section, nous présentons une liste de 8 attaques qui couvrent les vulnérabilitées reportées dans [28, 29]. Notre évaluation dans la section 4 montre que ces attaques peuvent corrompre, bloquer ou terminer les plateformes OSGi non protégées.

Manque d'isolation.

Nous avons mentionné dans l'introduction que les objets Java *java.lang.Class*, *java.lang.String* et les variables *static* sont partagées entre bundles. Un bundle malicieux peut modifier ou prendre un verrou sur un de ces objets et ainsi interférer avec l'exécution d'autres bundles. Nous considérons deux attaques représentatives de ce problème :

- A1 - Modification de variables *static* : Tous les bundles partagent les variables *static*. Par conséquent, un bundle peut modifier une variable *static, public*, non-*final* définies par d'autres bundles, par la plateforme OSGi ou par les classes de base de la JVM. Par exemple, un bundle malicieux peut mettre une variable partagée à *null*, et ainsi empêcher une exécution normale d'autres bundles. Les bundles peuvent découvrir les variables *static* définies par d'autres bundles soit à la compilation, soit à l'exécution grâce à l'API Java de reflection.
- A2 - Méthode ou block *synchronized* : Un bundle peut prendre un verrou sur les objets Java *java.lang.-Class*, *java.lang.String* ou les variables *static*, ce qui peut éventuellement bloquer la plateforme.

Manque de gestion de ressources

Un bundle est implémenté en utilisant un class loader Java spécifique. Cependant, les JVMs ne font pas du comptage de ressources au niveau du class loader. Dans le cas d'une utilisation excessive de ressources, il est impossible d'identifier le bundle fautif et arrêter son exécution. Compter les ressources par bundle pemettrait de détecter les attaques de déni de service suivantes :

- A3 - Epuisement de la mémoire : Un bundle malicieux peut consommer toute la mémoire disponible en maintenant des références vers beaucoup d'objets et de gros objets. Si les autres bundles essaient d'allouer de la mémoire, ils obtiennent une exception *OutOfMemoryError*.
- A4 - Boucle infinie : Un bundle malicieux consomme tout le temps CPU en exécutant une boucle infinie.
- A5 - Création excessive d'objets : Un bundle malicieux alloue en permanence des objets sans les référencer par la suite, et provoque l'activation du collecteur de mémoire et la finalisation d'objets, ce qui monopolise le CPU.
- A6 - Création excessive de threads : un bundle malicieux arrête la plateforme en excédant le nombre de threads supportés.
- A7 - Thread bloqué : Un bundle malicieux bloque lorsqu'il est appelé et ainsi empêche le thread de retourner à l'appelant.

Terminaison de bundles

Dans certains cas, la JVM est incapable de décharger un bundle et libérer toutes les ressources mémoire qu'il consomme. Par exemple, d'autres bundles peuvent référencer le bundle et ainsi empêcher la JVM de décharger les classes du bundle, parce qu'un appel vers une méthode de ces classes peut toujours arriver. Aussi, si la plateforme OSGi reconnait un bundle comme étant malicieux et souhaite le terminer, le fait que des méthodes définies par ce bundle sont en train d'être exécutées ou sont dans la pile d'exécution de threads l'en empêche.

- A8 - Pas de support pour la terminaison : Un bundle malicieux continue son exécution alors que la plateforme OSGi essaye de le décharger.

Enfin, une mauvaise utilisation des méthodes *finalize* est une vulnérabilité non reportée dans [28]. Cette vulnérabilité est présente dans toutes les plateformes qui effectuent du chargement dynamique de classes : un *finalize* malicieux peut entrer dans une boucle infinie et ainsi empêcher l'exécution d'autres *finalize*, nécessaires pour détruire un objet. Nous ne discutons pas de cette vulnérabilité dans cet article parce que nous considérons que la JVM ne devrait pas permettre à une classe non sûre de définir sa méthode *finalize*.

3. Conception et Implémentation d'I-JVM

L'objectif d'I-JVM est d'offrir l'isolation entre bundles tout en conservant le modèle de communication d'OSGi, qui se base sur des appels de méthode directs. Une application OSGi est constituée d'un ensemble de bundles chargés dynamiquement et de la plateforme OSGi elle-même. Pour ajouter de l'iso-

lation entre bundles, chaque bundle est exécuté dans un isolate. La plateforme OSGi s'exécute dans un isolate spécifique, Isolate0, qui a des droits priviliégiés.
Dans cette section, nous présentons I-JVM en détail. Nous nous concentrons sur trois aspects d'I-JVM : isolation, gestion de ressources et terminaison d'isolates. Nous décrivons ensuite l'implémentation d'I-JVM dans VMKit [15, 16] puis comment lancer la palteforme OSGi sur I-JVM.

3.1. Conception d'I-JVM

I-JVM permet un partage explicite d'objets entre isolates en passant des objets en paramètre d'un appel inter-isolate. Le point clé d'I-JVM est la migration de threads entre isolates dans un espace d'adressage unique, ce qui est nécessaire pour le partage d'objets. Plusieurs isolates s'exécutent dans le même contexte d'exécution, ce qui complexifie l'isolation, la gestion des ressources et la terminaison d'isolates.

Isolation et Migration de Threads

I-JVM exécute les isolates dans un espace d'adressage unique. Les isolates offrent un mécanisme de protection que nous appelons *léger*, pour que le code qui s'exécute dans un isolate n'interfère pas avec le code d'autres isolates. Dans I-JVM, un isolate est associé à un class loader Java, et englobe donc toutes les méthodes des classes chargées par ce class loader. L'isolate Isolate0 a des droits plus privilégiés que les autres isolates : il peut créer d'autres isolates, terminer un isolate ou terminer la plateforme. Le premier class loader Java créé par la JVM s'exécute dans Isolate0. Les class loaders suivants s'exécutent dans des isolates standards.
Un appel inter-isolate provoque une migration de thread. Chaque thread possède une référence à l'isolate dans lequel il est en train de s'exécuter. Cette référence est mise à jour à chaque appel inter-isolate. Une méthode définie dans une classe de base Java n'est pas exécutée dans un isolate particulier mais dans l'isolate qui l'a appelée.
Pour implémenter les isolates, le principal changement dans la spécification JVM est que chaque isolate possède sa propre copie des variables *static* et des objets *java.lang.Class* et *java.lang.String*. Ce changement est suffisant pour assurer l'isolation mémoire entre isolates car : (i) grâce au langage Java, un isolate ne peut pas construire une référence vers un objet d'un autre isolate (au contraire de langages comme C) et (ii) un isolate ne peut pas accéder aux champs ou méthodes privés d'un isolate grâce au mot clé *private*.
L'isolation des variables *static* est effectuée en associant un tableau de *Task Class Mirrors (TCM)* (défini par MVM dans [9]) à chaque classe. Le TCM d'une classe contient le statut de la classe (initialisée ou pas), les variables *static* et l'objet Java *java.lang.Class* associé. Chaque isolate possède un identifiant qui est un index dans le TCM. I-JVM utilise la référence d'isolate courante du thread pour indexer le TCM. Accéder à une variable *static* nécessite donc de récupérer l'environnement d'exécution du thread, récupérer l'isolate courant de cet environnement, lire le TCM et enfin lire la variable *static*. Comparé à une simple lecture de la variable *static* dans les JVM standard, le TCM nécessite deux lectures mémoire additionnelles. Un isolate doit aussi toujours vérifier le statut d'une classe avant de l'utiliser. Comme dans MVM, le compilateur ne peut pas enlever toutes ces vérifications.

Gestion des ressources

I-JVM contrôle l'exécution des isolates. Les ressources consommées par les méthodes d'un isolate sont comptabilisées à l'isolate. Les ressources consommées par les méthodes des classes de base de la JVM sont comptabilisées à l'isolate appelant. I-JVM comptabilise pour chaque isolate le temps CPU, la quantité mémoire utilisée, le nombre de threads créés, le nombre de connexions d'Entrées/Sorties (E/S) utilisées, la quantité d'octets lus ou écrits à travers une connexion E/S, et le nombre d'activations du collecteur de mémoire.

Mémoire et connexions : Le problème de la gestion de la mémoire dans I-JVM est que plusieurs isolates peuvent partager des objets. I-JVM suit le même algorithme de comptage de ressource mémoire défini par Price *et al.* [30], qui gère des domaines de protection distincts qui peuvent partager des objets. L'idée principale de l'algorithme est que c'est le collecteur mémoire qui effectue le comptage de ressources mémoires consommées.
Lorsque le collecteur mémoire s'exécute, il scanne la mémoire utilisée par les threads. Chaque objet est chargé au premier isolate qui le référence. Aux prochaines exécutions du collecteur, un objet partagé

pourra être chargé à un autre isolate. Même s'il est imprécis, cet algorithme offre une estimation qui est entre un minimum (objets uniquement atteignables par l'isolate) et un maximum (tous les objets référencés par l'isolate). Price *et al* concèdent que l'approche ouvre une porte à un *cheval de Troie* pour ressources [20] : un isolate alloue un objet conséquent qui devient partagé, mais le collecteur charge l'objet à un autre isolate. Nous discutons de ce problème dans la section 6.

Threads : I-JVM compte le nombre de threads qu'un isolate crée. Ils sont comptabilisés uniquement au créateur, même s'ils exécutent du code d'un autre isolate à travers un appel inter-isolate.

Lectures et écritures d'E/S : Une lecture ou une écriture sur une connexion d'E/S est instrumentée pour charger l'isolate qui effectue l'opération. L'approche est similaire à JRes [10] : peu de classes effectuent des lectures et écritures d'E/S dans les classes de base (qui sont les seules à pouvoir effectuer cette opération), ce qui rend l'implémentation relativement simple.

Temps CPU : Pour le comptage de CPU, nous avons étudié deux solutions. La première est d'ajouter du code à chaque appel inter-bundle pour mettre à jour le temps CPU. Le problème de cette approche est qu'un appel inter-bundle devient beaucoup plus coûteux parce qu'elle nécessite deux appels systèmes pour récupérer le temps courant, un au début et un à la fin de l'appel, ainsi que l'acquisition d'un verrou pour mettre à jour le temps CPU de l'isolate. Par conséquent, nous avons opté pour une seconde solution : I-JVM compte le temps CPU en inspectant régulièrement l'isolate courant de chaque thread qui s'exécute.

Collecteur mémoire : I-JVM utilise un seul collecteur pour tous les isolates. Pour détecter les attaques sur le collecteur, I-JVM compte le nombre de fois où un isolate provoque l'exécution du collecteur.

Terminaison d'isolate

Il y a deux principaux problèmes à résoudre pour terminer un isolate. Premièrement, les threads migrent entre isolates, donc I-JVM ne peut pas juste tuer les threads créés par l'isolate. Aussi, un thread créé par un autre isolate peut être en train d'exécuter du code de l'isolate qui doit se terminer. Deuxièmement, les objets alloués par l'isolate mais référencés par d'autres isolates ne peuvent pas être détruits.

Lorsque I-JVM termine un isolate, il arrête l'exécution de tous les threads en leur envoyant un signal. Le handler de ce signal est défini au démarrage d'I-JVM et ne peut pas être modifié par les isolates. Lorsqu'un thread reçoit le signal, il regarde et modifie sa pile d'exécution comme suit. Pour chaque méthode de la pile, si la méthode est appelée par une méthode qui appartient à l'isolate qui se termine, le thread change le pointeur de retour pour lancer une exception *StoppedIsolateException*. Le gestionnaire d'exceptions libère les verrous qui ont pu être pris par l'isolate qui se termine. L'isolate qui se termine ne peut pas attraper cette exception : s'il essaye de l'attraper dans du code Java, I-JVM l'ignore. Enfin, I-JVM effectue une action spéciale pour la méthode en train d'être exécutée :

– Si elle appartient aux classes de base de la JVM, I-JVM place le drapeau *interrupted* du thread. Cette approche est similaire à la terminaison de domaines de protection dans Spring [18].
– Si elle appartient à l'isolate qui termine, le thread lance une exception *StoppedIsolateException*.

Enfin, il ne faut plus que les méthodes des classes définies dans l'isolate soient exécutées. Pour ce faire, I-JVM change son comportement de deux manières : (i) il ne compile plus les méthodes non-compilées de l'isolate mais renvoie une exception *StoppedIsolateException* si un autre isolate essaye de les appeler et (ii) il insère un branchement au début de chaque méthode compilée définie par l'isolate. Ce branchement saute vers l'exécution d'une méthode qui lance une exception *StoppedIsolateException*.

Un isolate est définitivement détruit et la mémoire qu'il utilise libérée lorsqu'il n'existe plus d'objets dont la classe est définie par l'isolate.

Cette approche est similaire au mécanisme du *Thread.stop* de Java qui est maintenant dépréciée. Cette méthode est dépréciée parce qu'elle pouvait rendre des objets dans un état inconsistent. Avec notre approche, uniquement les objets référencés par l'isolate (qui peuvent être partagés) peuvent devenir dans un état inconsistent. Nous pensons que les avantages de la terminaison d'isolates l'emportent sur les problèmes d'inconsistence d'objets.

3.2. Implémentation d'I-JVM

Nous avons implémenté I-JVM avec VMKit [15, 16], une implémentation de la spécification JVM. En tout, I-JVM ajoute 650 lignes de code à VMKit, distribuées comme suit :

– Variables *static*, et objets *java.lang.Class* et *java.lang.String* : 200 lignes de code pour implémenter le TCM dans les classes. Les modifications sont à deux niveaux : (i) dans la représentation interne des classes Java, pour contenir le TCM et (ii) dans le traducteur de bytecode Java vers le compilateur pour modifier l'accès aux variables *static* et aux objets *java.lang.Class* qui se font désormais à travers le TCM ;
– Appel de méthode : 150 lignes de code pour mettre à jour la référence isolate du thread courant dans chaque méthode Java compilée par VMKit ;
– Comptage de ressources : 100 lignes de code pour le comptage de ressources mémoire, CPU, threads et E/S ;
– Un isolate pour un bundle : 50 lignes de code pour créer et associer un nouvel isolate à un class loader Java, lorsque celui-ci charge pour la première fois une classe ;
– Terminaison d'isolates : 150 lignes de code pour la terminaison d'isolates.

Ces nombres ne sont pas élevés et suggèrent qu'implémenter I-JVM dans d'autres JVM devrait être possible.

3.3. Exécuter OSGi sur I-JVM

Une application OSGi est composée d'un ensemble de bundles chargés dynamiquement et de la plateforme OSGi. Pour implémenter l'isolation, chaque bundle s'exécute dans son propre isolate ; la plateforme OSGi s'exécute dans l'isolate Isolate0.

Isolate0 est associé au class loader Java applicatif qui charge la fonction *main* de la plateforme OSGi. Lorsque OSGi charge un nouveau bundle, il alloue un nouveau class loader Java et I-JVM associe un isolate à ce class loader. Il est donc possible d'exécuter sans modification des plateformes OSGi existantes. La méthode *start* d'un bundle reçoit en paramètre un objet qui représente la plateforme OSGi. Cet objet est le premier objet partagé entre les bundles. Il est utilisé par les bundles pour enregistrer d'autres objets dans un service de nommage et pour trouver des références d'autres bundles. Par conséquent, au démarrage, un bundle ne peut accéder qu'aux objets présents dans ce service de nommage.

Nous définissons trois règles qu'une plateforme OSGi devrait prendre en compte pour plus de sécurité :

1. La plateforme devrait créer un nouveau thread à chaque fois qu'elle appelle les méthodes *start* et *stop* de bundles, de manière à éviter une exécution malicieuse d'un bundle qui pourrait bloquer la plateforme.

2. La plateforme devrait utiliser les permissions Java pour empêcher l'accès par les bundles à des ressources privilégiées. Par exemple, la JVM vérifie déjà qu'une classe a les droits nécessaires pour invoquer du code natif, soit par JNI, soit par *Runtime.exec*. De même pour l'appel *System.exit* qui arrête l'exécution de tout le système.

3. La plateforme devrait envoyer un événement *StoppedBundleEvent* à tous les bundles lorsqu'un bundle est en train d'être terminé. Les bundles peuvent alors agir en conséquence : un bundle peut ignorer l'événement ou supprimer les références qu'il a sur les objets alloués par le bundle qui se termine. Si le bundle ne supprime pas ces références, I-JVM pourra le lui charger. Les ressources du bundle qui se termine pourront être libérées uniquement lorsque tous les autres bundles auront laché leurs références vers elles.

Nous définissons aussi une règle qui devrait être suivie par les bundles. Chaque bundle devrait être préparé à attraper une exception lorsqu'il appelle un autre bundle, pour s'assurer qu'il va continuer son exécution même en présence d'un bundle malicieux. Comme avec toute JVM, I-JVM utilise les exceptions pour signaler une erreur dans l'exécution de code Java. Ces exceptions peuvent être des exceptions telles que *NullPointerException*, mais aussi des exceptions spécifiques à I-JVM lorsqu'un isolate est tué.

4. Evaluation

Dans cette section, nous commençons par motiver le besoin d'appels inter-bundles performants dans OSGi en analysant une application démo d'OSGi. Ensuite, nous évaluons les performances et l'occupation mémoire d'I-JVM, ainsi que sa robustesse contre les attaques.

4.1. Appels inter-bundles

Pour motiver l'approche OSGi et son besoin d'appels inter-bundles performants, nous évaluons l'application démo disponible sur le site de l'implémentation OSGi Felix [1]. L'application est de type *paint*,

Appel local	Appel RMI	Incommunicado	I-JVM
20µs	90ms	9ms	24µs

TAB. 1: Coût de 200 appels inter-bundle, selon le type de communication. Ces mesures ont été effectuées sur un Pentium D 3.0GHz avec 3Go de mémoire.

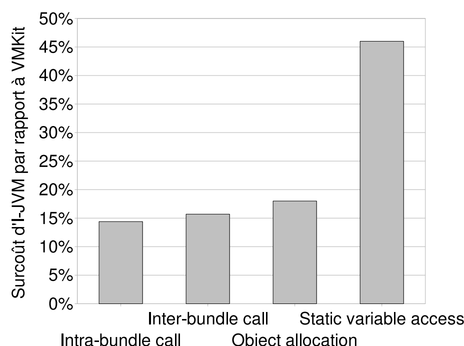

FIG. 1: Surcoût d'I-JVM pour les micro-benchmarks, relatif à VMKit.

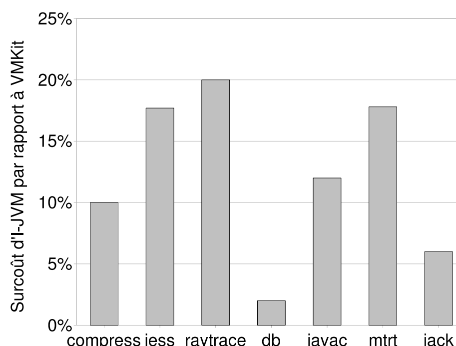

FIG. 2: Surcoût d'I-JVM pour les benchmarks de SpecJVM98, relatif à VMKit.

un logiciel pour faire des dessins, architecturée en bundles. Le tableau et les formes qui peuvent être dessinées sur le tableau sont implémentées en bundles. Un utilisateur peut ajouter des formes dynamiquement ou en supprimer grâce au gestionnaire de bundles OSGi.

Nous mesurons le nombre d'appels inter-bundles lorsque I-JVM exécute l'application. Placer une forme sur le tableau nécessite un appel inter-bundle. Bouger une forme de haut en bas et de gauche à droite du tableau nécessite environ 200 appels inter-bundles. La table 1 montre le temps pour effectuer ces 200 appels inter-bundles en fonction de l'implémentation de la communication. Nous évaluons quatre types d'implémentation : (i) appel local, (ii) appel RMI, le moyen standard pour effectuer des appels inter-applications, (iii) Incommunicado [27], l'implémentation des communications dans la MVM et (iv) I-JVM. Nous n'avons pas accès au code de Incommunicado, donc notre évaluation se base sur les résultats donnés dans [27], où Incommunicado est 10 fois plus performant que RMI. Comme les isolates s'exécutent sur une même machine, Incommunicado court-circuite tous les mécanismes réseaux coûteux de RMI. I-JVM a de meilleures performances que les autres approches pour les appels inter-isolates.

4.2. Performances d'I-JVM

VMKit est un prototype de recherche dont les performances sont inférieures aux JVMs industrielles [15, 16]. Nous reportons donc les performances d'I-JVM relatives à VMKit. Toutes les expériences ont été effectuées sur un Pentium D 3.0GHz avec 3Go de mémoire sous Linux Mandriva 2.6.23.

Pour évaluer le surcoût de l'isolation et de la gestion des ressources dans I-JVM, nous avons effectué une série de micro-benchmarks : appels intra-isolates et inter-isolates, allocations d'objets et accès à une variable *static*. Nous avons mesuré le surcoût en effectuant la même opération un million de fois. Nous avons aussi executé le benchmark SpecJVM98 dans un isolate pour mesurer le coût d'une exécution normale dans I-JVM. Enfin, nous avons mesuré le surcoût mémoire d'I-JVM avec deux implémentations d'OSGi : Felix [1] de la fondation Apache et Equinox [2] du projet Eclipse.

La figure 1 montre le surcoût d'I-JVM relatif à VMKit pour les micro-benchmarks. I-JVM ajoute quatre lectures et deux instructions de comparaison lors d'un appel intra-isolate pour vérifier si l'appel n'est pas un appel inter-isolate. Pour un appel inter-isolate, I-JVM met à jour l'isolate courant du thread et rajoute ainsi quatre nouvelles écritures. (deux à l'entrée de la méthode pour l'identifiant et la référence et deux en sortie). Au final, un appel intra-isolate a un surcoût de 14% et un appel inter-isolate a un

surcoût de 16%.
Nous avons aussi étudié les performances pour allouer un objet. Dans VMKit et I-JVM la taille d'un objet de base est 28 octets. Les résultats montrent qu'il y a un surcoût de 18% comparé à VMKit, dû au comptage et à la gestion des resources, ainsi que le coût d'un appel intra-bundle en appelant le constructeur de *java.lang.Object*. Enfin, nous avons mesuré l'accès à une variable *static*. Le benchmark montre que le surcoût est de 46% comparé à VMKit, dû à l'utilisation du TCM et au test d'initialisation de la classe (pour vérifier que la variable *static* a bien été allouée). En appliquant des optimisations au niveau du compilateur, le surcoût d'accéder à une variable *static* un million de fois n'est plus que de 1% car les instructions supplémentaires ne sont exécutées qu'une seule fois.
Nous avons mesuré le temps d'exécution des benchmarks SpecJVM98 [4] lorsqu'ils s'exécutent dans un isolate. La figure 2 montre que le surcoût d'I-JVM est en-dessous de 20% pour tous les benchmarks. En comparaison, les résultats pour MVM sont au maximum de 10% dans MVM selon [9]. La raison principale de cette différence est qu'I-JVM ajoute un surcoût dû à la gestion des ressources, et effectue quatre lectures et deux comparaisons en plus à chaque appel intra-bundle. Cependant le coût d'un appel inter-isolate est bien plus conséquent dans MVM que dans I-JVM [27].

Enfin, nous avons mesuré le surcoût mémoire d'I-JVM avec deux implémentations OSGi. I-JVM demande plus de mémoire que VMKit à trois endroits : (i) le TCM pour chaque classe, (ii) un ensemble d'objects *java.lang.String* par isolate et (iii) une suite de statistiques pour la gestion des ressources. La figure 3 montre la mémoire utilisée lorsque Felix et Equinox s'exécutent avec la configuration de base. Felix exécute la plateforme OSGi ainsi que trois bundles (administration, shell, service de nommage) et Equinox exécute la plateforme OSGi et 22 bundles. Au final, le surcoût mémoire d'I-JVM par rapport à VMKit pour les deux implémentations d'OSGi est en-dessous de 16%.

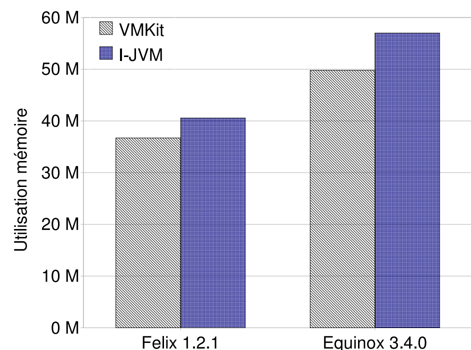

FIG. 3: Utilisation mémoire d'I-JVM et VMKit pour deux implémentations OSGi.

4.3. Robustesse contre les attaques

Dans la section 2, nous avons présenté un ensemble d'attaques qui testent la robustesse d'une implémentation de JVM qui exécute une plateforme OSGi. Dans cette section, nous comparons les résultats entre I-JVM et la JVM de Sun, OpenJDK 6, lorsque celles-ci exécutent ces attaques. I-JVM permet d'empêcher les huit attaques que nous avons identifeées avec l'aide d'un administrateur. Avec la JVM de Sun, un administrateur perd le controle de sa plateforme et n'a pas la possibilité d'arrêter les attaques, même s'il connait les bundles malicieux. Par conséquent, un redémarrage de la plateforme est la seule solution pour la JVM de Sun.

A1 - Modification de variables *static*

Un bundle A définit un tableau comme variable *static* et travaille sur les éléments de ce tableau. Le bundle B localise ce tableau (soit à la compilation, soit à travers l'API de refection Java). B modifie le contenu du tableau en mettant *null* dans le tableau.
`Résultat avec la JVM de Sun :` Bundle A lance une *NullPointerException*.
`Résultat avecI-JVM :` I-JVM isole les bundles pour qu'ils ne puissent pas avoir accès aux variables *static* des autres. Le tableau est dupliqué, donc les modifications effectuées par B se font sur la copie locale du tableau chez B. Le bundle A continue à travailler sans erreur sur le tableau.

A2 - Méthode ou block *synchronized*.

Le bundle A appelle une méthode *static, synchronize* définie par lui-même. Le bundle acquiert donc un verrou sur un objet Java *java.lang.Class* qui représente la classe. Un autre bundle B fait un *synchronize* explicite sur le même objet et ne lache jamais le verrou.
`Résultat avec la JVM de Sun` : Le bundle A est bloqué.
`Résultat avec I-JVM` : I-JVM ne permet pas de partager les objets *java.lang.Class*, *java.lang.String* et les variables *static*, donc il n'y a pas d'interférence entre les bundles qui ne communiquent pas entre eux. Le bundle A continue son exécution.

A3 - Epuisement de la mémoire.

Un ensemble de bundles s'exécute sur la plateforme. La plateforme OSGi installe dynamiquement un nouveau bundle qui alloue beaucoup d'objets et les place dans un tableau, empêchant donc le collecteur de mémoire de supprimer ces objets.
`Résultat avec la JVM de Sun` : Les bundles lancent une *OutOfMemoryError* lorsqu'ils veulent allouer un objet.
`Résultat avec I-JVM` : I-JVM compte la mémoire utilisée par chaque bundle. Sur cette information, l'administrateur tue le bundle malicieux et tous les autres bundles continuent de s'exécuter.

A4 - Boucle infinie.

Un ensemble de bundles s'exécute sur la plateforme. La plateforme OSGi installe dynamiquement un nouveau bundle qui effectue une boucle infinie.
`Résultat avec la JVM de Sun` : Les bundles non malicieux s'exécutent lentement.
`Résultat avec I-JVM` : I-JVM compte le temps CPU utilisé par chaque bundle. Sur cette information, l'administrateur tue le bundle malicieux et tous les autres bundles continuent de s'exécuter.

A5 - Création excessive d'objets.

Un ensemble de bundles s'exécute sur la plateforme. La plateforme OSGi installe dynamiquement un nouveau bundle qui alloue plusieurs objets mais ne maintient pas de références vers eux. Le collecteur de mémoire est donc activé plusieurs fois.
`Résultat avec la JVM de Sun` : La JVM passe son temps dans le collecteur mémoire. Les bundles non-malicieux s'exécutent lentement.
`Résultat avec I-JVM` : I-JVM compte le nombre de fois qu'un bundle active le collecteur mémoire. Sur cette information, l'administrateur tue le bundle malicieux et les autres bundles continuent de s'exécuter.

A6 - Création excessive de threads.

Un ensemble de bundles s'exécute sur la plateforme. La plateforme OSGi installe dynamiquement un nouveau bundle qui crée indéfiniment des threads.
`Résultat avec la JVM de Sun` : Les bundles lancent une *OutOfMemoryError* lorsqu'ils veulent allouer un objet ou un nouveau thread.
`Résultat avec I-JVM` : I-JVM compte le nombre de threads créés par chaque bundle. Sur cette information, l'administrateur tue le bundle malicieux et tous les autres bundles continuent de s'exécuter.

A7 - Thread bloqué.

Un bundle A appelle une méthode d'un bundle B. Le bundle B appelle alors *Thread.sleep(0)*.
`Résultat avec la JVM de Sun` : L'exécution ne revient jamais vers le bundle A.
`Résultat avec I-JVM` : I-JVM inspecte le bundle courant de chaque thread et compte le nombre de threads endormis dans le bundle. Sur cette information, l'administrateur tue le bundle qui appelle *Thread.sleep*. Si le bundle A a été écrit pour attraper l'exception *Stopped-Isolate-Exception*, l'exécution retourne à A. Sinon, l'exception est attrapée à un niveau plus bas de la pile d'exécution.

A8 - Pas de support pour la terminaison.

Un bundle A appelle un bundle B et attend une valeur de retour. Le bundle B retourne un objet qui référence la représentation OSGi interne du bundle. Le bundle A écrit cette référence dans une de ses variables et ainsi empêche la terminaison du bundle B. Le bundle B effectue une attaque de déni de service et l'administrateur souhaite le terminer.
`Résultat avec la JVM de Sun :` La plateforme OSGi ne peut pas décharger le bundle, et l'attaque continue de s'exécuter.
`Résultat avec I-JVM :` Tous les threads qui exécutent du code du bundle B lancent une exception et l'exécution ne revient jamais au bundle B.

Ces résultats montrent qu'I-JVM offre une meilleure isolation ainsi qu'une meilleure gestion des ressources comparé à une JVM standard. En isolant les bundles et en comptant les ressources, I-JVM permet à un administrateur de trouver et terminer les bundles malicieux. L'isolation et la terminaison n'ont pas besoin en soi de l'assistance d'un administrateur. Cependant, le fait de trouver le bundle malicieux est toujours à la charge de l'administrateur. Nous prévoyons pour des travaux futurs, d'écrire des politiques de ressources par bundle ou par groupe de bundles pour automatiser la détection d'attaques de déni de service.

5. Travaux connexes

Notre approche prend ses idées de plusieurs domaines de recherche. Dans cette section, nous décrivons comment notre travail se compare avec (i) les systèmes d'exploitation et (ii) les JVMs à gestion de ressources et isolation.

5.1. Systèmes d'exploitation

L'architecture OSGi est très similaire à celle des micro-noyaux, où plusieurs modules s'exécutent dans leur propre domaine de protection et communiquent au travers d'IPCs (Inter-Process Communications). Cependant, les domaines de protection dans les micro-noyaux sont implémentés en espace d'adressage virtuel multiple et les communications doivent donc effectuer des copies d'objets entre deux espaces d'adressage. Beaucoup de projets se sont concentrés sur l'optimisation des IPCs dans les micro-noyaux [7, 11, 13, 25], mais les performances sont toujours largement inférieures à un appel local direct.
Les micro-noyaux tels que JX [17] et Singularity [21] sont développés dans un langage sûr de manière à exécuter plusieurs modules dans un seul et même espace d'adressage. Ces systèmes isolent les modules en implémentant des communications au travers d'objets dits *capacités* [23] ou des tas partagés [12]. Dans les deux cas, ces approches ne sont pas compatibles avec les applications OSGi existantes.
Les systèmes d'exploitation Scout [33], Rialto [22], ou les *conteneurs de ressources* [6] définissent de nouvelles approches pour la gestion des ressources. Les threads dans ces systèmes ne sont pas confinés à un unique domaine de protection, mais migrent entre les domaines. Le système charge les ressources à une *entité d'exécution* (autrement dit, un utilisateur), et un thread est confiné à un utilisateur. Ces systèmes font la différence entre un utilisateur et un domaine de protection. Dans OSGi, un domaine de protection (i.e. un bundle) *est* un utilisateur.

5.2. Isolation dans les JVMs

Le mécanisme d'isolation standard dans les JVMs est basé sur les class loaders Java [24]. Les class loaders offrent une isolation de nommage : il n'y a pas de collision de noms lorsque deux classes avec le même nom sont chargées par deux class loaders différents. Les class loaders ont des garanties faibles en isolation, parce qu'ils permettent le partage des variables *static*, *java.lang.String* et *java.lang.Class* entre class loaders. OSGi par défaut utilise les class loaders et les permissions Java pour limiter les droits d'un bundle. Une méthode qui effectue une opération privilégiée, telle que *System.exit* demande à la JVM les droits de la méthode qui l'a appelée. La JVM inspecte la pile d'exécution pour trouver quel est le class loader qui a chargé la méthode. Un class loader peut être attaché à une politique de sécurité, qui indique l'ensemble des droits qu'il possède et que ses méthodes peuvent exécuter. L'inspection d'une pile est une opération coûteuse, qui réduirait dramatiquement les performances de notre approche s'il fallait l'exécuter à chaque accès d'une variable *static* ou d'objets *java.lang.Class* ou *java.lang.String*.

Il existe beaucoup de projets qui ajoutent la notion de processus logiciels dans la JVM, en opposition aux processus matériels où l'espace d'adressage virtuel permet d'isoler les processus [5, 9, 10, 19]. Ces sytèmes proposent isolation, terminaison et gestion des ressources, mais change le modèle de programmation et de communication de Java. Par conséquent, ils ne sont pas compatibles avec les applications OSGi existantes.

La Secure JVM est très proche de notre travail [35]. C'est une implémentation de JVM au dessus d'un système d'exploitation extensible. Les modules s'exécutent dans leur propre espace d'adressage. Les communications inter-modules s'écrivent comme des appels de procédures locaux, mais le système effectue un remapping de pages et la copie d'objets lors d'un appel inter-module, ce qui a un surcoût par rapport à un appel local. De plus, la Secure JVM ne s'exécute pas sur d'autres systèmes d'exploitation.

Rudys *et al.* definissent la *terminaison légère* pour les sytèmes orientés langage [32]. L'approche consiste à ajouter dynamiquement à chaque classe Java une variable *static* qui indique si la classe doit être arrêtée. La variable est vérifiée régulièrement dans les méthodes définies dans la classe. Le code pour vérifier est inséré sur les branchements de retour de boucle et après les appels de méthode. Rudys *et al.* ont implémenté leur système en faisant de la réécriture dynamique de bytecode. I-JVM implémente une sorte de *terminaison dure* dans la JVM : la JVM modifie les piles d'exécution des threads pour certifier que le code d'une classe ne sera plus jamais exécuté.

Au final, I-JVM est une combinaison de trois projets de recherche : les isolates dans Java [9], la terminaison légère [32] et le comptage de ressource mémoire [30] dans les systèmes orientés langage.

Enfin, dans le contexte d'OSGi, Service Coroner [14] est un outil d'administration pour trouver les bundles en état zombie. Un bundle en état zombie est un bundle terminé qui ne peut pas être déchargé car d'autres bundles maintiennent des références vers le bundle. Service Coroner ne modifie pas la JVM ou une implémentation d'OSGi, mais utilise la programmation orientée aspect pour instrumenter la création de bundles. I-JVM et Service Coroner s'attaquent à deux différents problèmes dans OSGi mais ont tous deux pour but de donner plus de controle sur la plateforme OSGi à l'administrateur.

6. Conclusion et Travaux Futurs

Nous avons décrit la conception et l'implémentation d'I-JVM, une machine virtuelle Java étendue avec l'isolation et la terminaison de composants. I-JVM offre une isolation dite *légère* de bundles OSGi tout en conservant des communications performantes grâce à la migration de threads entre bundles et le partage direct d'objets. L'architecture d'isolate d'I-JVM permet de faire de la gestion de ressources par bundle, ce qui peut être utilisé par un administrateur pour localiser et terminer les bundles malicieux. Même si l'isolation et la gestion des ressources entrainent un surcoût à l'exécution comparé à d'autres JVMs standard, notre évaluation montre que I-JVM est capable d'informer un administrateur d'une attaque de déni de service, ce que ne peuvent pas faire d'autres JVMs.

Dans cet article, nous avons considéré la gestion des ressources comme une aide pour un administrateur pour trouver les problèmes de ressources et tuer les bundles qu'il pense être malicieux. Cependant, l'administrateur doit être conscient de ce que I-JVM mesure : comme indiqué par Price *et al.* [30], l'utilisation mémoire enregistrée pour un bundle est située entre un maximum (tous les objets référencés par le bundle lui sont comptabilisés) et un minimum (les objets référencés uniquement par le bundle lui sont comptabilisés). Les bundles ont donc la possibilité de créer un cheval de Troie au niveau ressource [20]. Les politiques de ressource et les bundles doivent donc être écrits en sachant que ces problèmes peuvent arriver. De la même manière, notre comptage de temps CPU n'est pas aussi précis que dans des systèmes ou les threads ne peuvent pas migrer. Des techniques plus précises avec *profiling* [8] ont un surcoût conséquent à l'exécution, entre 67% et 279%, que nous ne pouvons pas nous offrir.

Nous prévoyons en travaux futurs d'implémenter des politiques de sécurité qui pourront aider la détection de ces problèmes de ressources. En plus des capacités de terminaison et d'isolation d'I-JVM présentées dans cet article, I-JVM deviendrait alors un système encore plus robuste pour exécuter des applications OSGi.

Téléchargement

I-JVM est placée sous une license libre et est téléchargeable à l'adresse : `http ://vmkit.llvm.org`.

Bibliographie

1. Apache felix. http ://felix.apache.org/site/index.html.
2. Equinox. http ://www.eclipse.org/equinox.
3. Jonas J2EE Server. http ://jonas.objectweb.org.
4. SPECjvm98. http ://www.spec.org/jvm98/.
5. G. Back, W. H. Hsieh, and J. Lepreau. Processes in KaffeOS : isolation, resource management, and sharing in Java. In *Proceedings of the Operating Systems Design and Implementation Symposium*, pages 333–346, San Diego, USA, October 2000. USENIX.
6. G. Banga, P. Druschel, and J. Mogul. Resource containers : a new facility for resource management in server systems. In *Proceedings of the Operating Systems Design and Implementation Symposium*, pages 45–58, New Orleans, USA, February 1999. USENIX.
7. B. Bershad, T. Anderson, E. Lazowska, and H. Levy. Lightweight remote procedure call. *Transactions on Computer Systems*, 8(1) :37–55, 1990.
8. W. Binder. Portable and accurate sampling profiling for Java. *Software : Practice and Experience*, 36(6) :615–650, 2006.
9. G. Czajkowski and L. Daynès. Multitasking without compromise : a virtual machine evolution. In *Proceedings of the Object Oriented Programming, Systems, Languages, and Applications Conference*, pages 125–138, Tampa Bay, USA, October 2001. ACM.
10. G. Czajkowski and T. Eicken. JRes : a resource accounting interface for Java. In *Proceedings of the Object Oriented Programming, Systems, Languages, and Applications Conference*, pages 21–35, Vancouver, Canada, October 1998. ACM.
11. R. Draves, B. Bershad, R. Rashid, and R. Dean. Using continuations to implement thread management and communication in operating systems. In *Proceedings of the Symposium on Operating Systems Principles*, pages 122–136, Pacific Grove, USA, October 1991. ACM.
12. M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. Hunt, J. Larus, and S. Levi. Language support for fast and reliable message-based communication in Singularity OS. In *Proceedings of the EuroSys Conference*, pages 177–190, Leuven, Belgium, April 2006. ACM.
13. B. Ford and J. Lepreau. Evolving Mach 3.0 to a migrating thread model. In *Proceedings of the USENIX Winter Technical Conference*, pages 97–114, San Francisco, USA, January 1994. USENIX.
14. K. Gama and D. Donsez. Using the service coroner tool for diagnosing stale references in the OSGi platform. In *Proceedings of the Middleware Conference Companion*, pages 58–61, Leuven, Belgium, December 2008. ACM.
15. N. Geoffray. The VMKit project. In *Proceedings of the LLVM Developer Meeting*, August 2008. http ://vmkit.llvm.org.
16. N. Geoffray, G. Thomas, C. Clément, and B. Folliot. A lazy developer approach : building a JVM with third party software. In *Proceedings of the Principles and Practice of Programming In Java Conference*, pages 73–82, Modena, Italy, September 2008. ACM.
17. M. Golm, M. Felsera, C. Wawersich, and J. Kleinoeder. The JX operating system. In *Proceedings of the Usenix Annual Technical Conference*, pages 45–58, Monterey, USA, June 2002. USENIX.
18. G. Hamilton and P. Kougiouris. The Spring Nucleus : a microkernel for objects. In *Proceedings of the USENIX Summer Technical Conference*, pages 1–15, Cincinnati, USA, June 1993. USENIX.
19. C. Hawblitzel, C. Chang, G. Czajkowski, D. Hu, and T. von Eicken. Implementing multiple protection domains in Java. In *Proceedings of the USENIX Annual Technical Conference*, pages 259–270, New Orleans, USA, June 1998. USENIX.
20. C. Hawblitzel and T. von Eicken. Luna : a flexible Java protection system. In *Proceedings of the Operating Systems Design and Implementation Symposium*, pages 391–401, Boston, USA, December 2002. ACM.
21. G. Hunt, M. Aiken, M. Fähndrich, C. Hawblitzel, O. Hodson, J. Larus, B. Steensgaard, D. Tarditi, and T. Wobber. Sealing OS processes to improve dependability and safety. In *Proceedings of the Eurosys Conference*, pages 341–354, Lisboa, Portugal, April 2007. ACM.
22. M. Jones, P. Leach, R. Draves, and J. Barrera. Modular real-time resource management in the Rialto operating system. In *Proceedings of the Hot Topics in Operating Systems Workshop*, pages 12–17, Washington, USA, May 1995. IEEE Computer Society.
23. H. Levy. *Capability-Based Computer Systems*. Butterworth-Heinemann, Newton, USA, 1984.
24. S. Liang and G. Bracha. Dynamic class loading in the Java virtual machine. In *Proceedings of the Object Oriented Programming, Systems, Languages, and Applications Conference*, pages 36–44, Vancouver, Canada, October 1998. ACM.
25. J. Liedtke, K. Elphinstone, S. Schiinberg, H. Hartig, G. Heiser, N. Islam, and T. Jaeger. Achieved IPC performance. In *Proceedings of the Hot Topics in Operating Systems Workshop*, pages 28–31, Washington, USA, May 1997. IEEE Computer Society.
26. OSGi Alliance. OSGi service platform, core specification release 4.1. Draft, 05 2007.
27. K. Palacz, J. Vitek, G. Czajkowski, and L. Daynès. Incommunicado : efficient communication for isolates. In *Proceedings of the Object-Oriented Programming, Systems, Languages, and Applications Conference*, pages 262–274, Seattle, USA, November 2002. ACM.
28. P. Parrend and S. Frénot. Classification of component vulnerabilities in Java service oriented programming platforms. In *Proceedings of the Component-Based Software Engineering Symposium*, pages 80–96, Karlsruhe, Germany, October 2008. Springer.
29. P. Parrend and S. Frénot. Security benchmarks of OSGi platforms : toward hardened OSGi. *Software : Practice and Experience*, 39(5) :471–499, April 2009.
30. D. Price, A. Rudys, and D. Wallach. Garbage collector memory accounting in language-based systems. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 263–274, Oakland, USA, May 2003. IEEE Computer Society.
31. Y. Royon and S. Frénot. Multiservice home gateways : business model, execution environment, management infrastructure. *IEEE Communications Magazine*, 45(10) :122–128, October 2007.
32. A. Rudys and D. Wallach. Termination in language-based systems. *ACM Transactions on Information and System Security*, 5(2) :138–168, 2002.
33. O. Spatscheck and L. Peterson. Defending against denial of service attacks in Scout. In *Proceedings of the Operating Systems Design and Implementation Symposium*, pages 59–72, New Orleans, USA, February 1999. USENIX.
34. E. Tilevich and Y. Smaragdakis. J-Orchestra : automatic Java application partitioning. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 178–204, Malaga, Spain, June 2002. Springer.
35. L. van Doorn. A secure Java virtual machine. In *Proceedings of the USENIX Security Symposium*, pages 19–34, Denver, USA, August 2000. USENIX.