HAL
open science

Éléments de sécurité dans l'architecture de systèmes répartis THINK.

Christophe Rippert, Jean-Bernard Stefani

To cite this version:

Christophe Rippert, Jean-Bernard Stefani. Éléments de sécurité dans l'architecture de systèmes répartis THINK.. Journées francophones des jeunes chercheurs en systèmes d'exploitation de l'ACM SIGOPS France, Apr 2002, Hammamet, Tunisie. hal-00310151

HAL Id: hal-00310151

https://hal.science/hal-00310151

Submitted on 8 Aug 2008

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

RENPAR'14 / ASF / SYMPA
Hammamet, Tunisie, 10 – 13 avril 2002

Éléments de sécurité dans l'architecture de systèmes répartis THINK

Christophe Rippert, Jean-Bernard Stefani

Projet SARDES - Laboratoire SIRAC (INPG - INRIA - UJF)
INRIA Rhône-Alpes,
655 avenue de l'Europe, Montbonnot St Martin,
38334 St Ismier Cedex - France
Christophe.Rippert@inria.fr

Résumé
Les architectures de noyaux monolithiques s'avérant inadéquates pour exploiter pleinement les ressources matérielles, on s'intéresse dans cet article au modèle exo-noyau et à l'architecture de systèmes répartis THINK, du point de vue de la sécurité et de la préservation de la qualité de service offerte par le système. On présente donc le travail d'analyse effectué concernant les ressources matérielles principales d'un ordinateur et les outils élémentaires à implémenter pour garantir une protection de ces ressources et de la qualité de service du système. On décrit aussi l'outil d'isolation logicielle des processus en cours d'implémentation et on donne un exemple concret d'utilisation des outils élémentaires proposés.

Mots-clés : exo-noyau, flexibilité, isolation, qualité de service, sécurité

---

1. Introduction

Les architectures de noyaux monolithiques ont prouvé leurs faiblesses sur le plan de la flexibilité d'adaptation des services qu'ils proposent au concepteur d'applications. Il apparaît donc intéressant de fournir des architectures alternatives, donnant au programmeur un meilleur contrôle sur les ressources matérielles dont il dispose et lui permettant d'exploiter pleinement leurs capacités. Les architectures micro-noyau s'étant avérées intéressantes mais incomplètes, on s'intéresse actuellement à l'architecture exo-noyau dont le jusqu'au-boutisme en matière d'exigences de flexibilité et de performances va dans le sens d'une utilisation optimale des ressources matérielles. Cependant, les performances et la flexibilité d'un système ne doivent en aucun cas être synonymes d'insécurité, et l'architecture exo-noyau, dont la caractéristique principale est de permettre aux applications d'accéder directement aux ressources matérielles, s'avère particulièrement vulnérable aux attaques et particulièrement celles cherchant à dégrader la qualité du service offert en monopolisant les ressources.

On présente dans cet article nos travaux concernant l'implémentation d'outils élémentaires de sécurité dans l'architecture de systèmes répartis THINK. On rappellera tout d'abord les différentes architectures de noyau existantes et on mettra en évidence l'intérêt de l'architecture exo-noyau, avant de présenter l'exo-noyau de THINK et son canevas logiciel. On détaillera ensuite l'analyse effectuée concernant la préservation de la qualité de service pour les ressources matérielles principales de l'ordinateur et on présentera l'implémentation en cours d'un outil élémentaire de sécurité assurant l'isolation logicielle des processus. On montrera enfin comment ces outils peuvent être facilement utilisés grâce à un exemple concret.

2. Architectures de noyau

Les noyaux de système d'exploitation classiques [1] fournissent au programmeur d'applications de nombreuses abstractions système (ex. gestionnaire de mémoire, système de fichiers, protocoles réseau, etc.) afin de faciliter le développement des applications. Cependant, ce type de noyaux (appelés noyaux monolithiques) est généralement considéré comme volumineux, lent et peu évolutif [2] par les développeurs de noyaux qui cherchent des solutions alternatives depuis plus de 30 ans [3].

2.1. Les micro-noyaux et noyaux extensibles

L'architecture micro-noyau [4] a été proposée pour améliorer la portabilité, la modularité et l'évolutivité des noyaux classiques. Un micro-noyau inclut les abstractions supposées nécessaires à toutes les applications, comme un gestionnaire de mémoire ou un ordonnanceur par exemple. Ce micro-noyau peut être étendu par des abstractions proposées à l'extérieur du noyau sous forme de serveurs. Les applications peuvent alors utiliser ces abstractions facultatives via un mécanisme d'appel de procédure distante classique. L'architecture micro-noyau offre une plus grande flexibilité qu'un noyau monolithique, puisqu'elle n'oblige pas le programmeur d'applications à utiliser des abstractions haut niveau dont il n'a pas besoin. Cette architecture a donnée naissance à de nombreux prototypes [5][6][7]. Elle a de plus inspiré les architectures dites de noyaux extensibles [8] et de composants de noyaux flexibles [9] qui fournissent au programmeur système des outils pour composer son noyau en fonction de ses besoins. Cependant, elle se révèle insuffisante pour les applications qui nécessitent un contrôle à grain fin des ressources matérielles, pour obtenir des performances optimales ou mettre en oeuvre des politiques de gestions spécifiques, car certaines abstractions (e.g le gestionnaire mémoire, l'ordonnanceur, etc) restent obligatoires.

2.2. L'exo-noyau

L'architecture exo-noyau [10][11] est basée sur l'idée qu'un noyau ne doit imposer aucune abstraction au programmeur d'applications, même les plus basiques. Un exo-noyau ne fournit donc que des interfaces permettant d'utiliser le matériel, sans ajouter de fonctionnalités supplémentaires. Des abstractions de plus haut niveau peuvent être fournies, mais leur utilisation doit rester optionnelle. Par exemple, un programmeur désirant utiliser un ordonnanceur pourra utiliser celui fournit par le système s'il le désire, mais rien ne doit l'empêcher de programmer et d'utiliser le sien si son application le nécessite. En fournissant ces abstractions optionnelles sous formes de bibliothèques, l'architecture permet au programmeur d'applications de se construire un système à la carte et de n'utiliser que les abstractions dont il a vraiment besoin, ce qui permet d'éviter de saturer le noyau (et donc la mémoire) avec des services inutiles. Ce type d'architectures s'adresse bien évidemment plus particulièrement aux concepteurs d'applications fortement liées au système d'exploitation ou s'exécutant sur des supports matériels dédiés à une application (comme certains systèmes embarqués par exemple). C'est à ce type de programmeur, à mi-chemin entre le programmeur système et le développeur d'applications, auxquels nous nous intéresserons dans la suite de cet article.

3. THINK

L'architecture de système réparti THINK [12] (THINK est l'abréviation de THink Is Not a Kernel) est notre proposition de plateforme pour le développement de noyaux de systèmes d'exploitation répartis. THINK fournit donc les interfaces nécessaires à la programmation du matériel sous-jacent, ainsi que de nombreuses abstractions optionnelles, conformément au modèle exo-noyau. Le développement sous THINK est facilité par la présence d'une architecture basée sur un modèle à objets. Dans THINK, toutes les ressources (logicielles et matérielles) sont en effet considérées comme des objets. Ces objets exportent des interfaces qui définissent leur comportement et les rendent accessibles aux autres objets. Chaque interface a un nom dans un contexte de nommage donné, et est liée à d'autres interfaces par des liaisons. Une liaison est fondamentalement un canal de communication entre des objets. Ces liaisons peuvent prendre de nombreuses formes, aussi simple que l'association entre un nom de variable et l'espace mémoire qui lui est alloué, ou plus complexe comme une connexion réseau entre des objets localisés sur des machines distantes. Les liaisons sont crées par des objets dédiés, appelés usines à liaisons, dont la fonction principale (i.e. créer une liaison entre l'objet appelant et une interface identifiée par son nom) peut être librement étendue afin d'assurer le comportement voulu. Enfin, les objets peuvent être groupés en domaines selon une propriété commune (ex. domaine de sécurité, domaine de tolérance aux pannes, etc.).

4. Outils élémentaires de sécurité

Le principal inconvénient de l'architecture exo-noyau est sa vulnérabilité aux attaques contre la sécurité du système. En effet, en fournissant un accès direct aux ressources matérielles aux applications, on favo-

rise les accès illégaux à des données pouvant être stratégiques pour le bon fonctionnement du système, ce qui est en général impossible dans un système d'exploitation classique. Cependant, les techniques de protection implémentées habituellement dans les systèmes d'exploitation se révèlent peu adaptées à la philosophie exo-noyau, car elles ont en général tendance à imposer une politique de gestion des ressources qui peut être contraire aux besoins du programmeur.

Le but de notre travail est donc de fournir les outils élémentaires nécessaires à l'implémentation de politiques de sécurité quelconques. Ces outils devront être indispensables (i.e. il n'est pas possible d'implémenter de politique de sécurité sans eux), non contraignants (i.e. ils ne doivent pas imposer ou gêner l'implémentation d'une politique de sécurité particulière) et bien évidemment le plus performants possible. Leurs buts est de permettre au programmeur de mettre en place la politique de sécurité qui lui semble la plus appropriée pour faire face aux menaces auxquelles son application pourrait être confrontée.

4.1. Ressources matérielles fondamentales

Pour identifier les outils élémentaires à fournir, on se base sur l'étude de quatre composants matériels essentiels :

Le processeur

Le partage équitable du processeur nécessite la gestion des interruptions afin de permettre la préemption des processus. Comme la plupart des processeurs fournissent un mécanisme de gestion des interruptions, il suffira de réifier ce mécanisme par des interfaces appropriées dans l'exo-noyau. Ces interfaces sont d'ores et déjà implémentées dans THINK. De plus, il est important de protéger les méthodes d'enregistrement et de modification de traitants d'interruption, pour éviter qu'un programme malicieux ne modifie un traitant en sa faveur. Dans THINK, cela peut être réalisé aisément en sécurisant la création de liaisons (par exemple avec un système d'identification à base de capacités) et en garantissant leur intégrité (grâce à un mécanisme de cryptage et de calcul de somme de contrôle par exemple).

La mémoire

L'implémentation d'une politique de protection, quelqu'elle soit, nécessite la notion d'isolation des processus. Comme on ne peut supposer que la machine sous-jacente fournisse au niveau matériel un tel mécanisme d'isolation (certains systèmes embarqués sont basés sur des processeurs très limités), il paraît nécessaire d'implémenter de façon logicielle ce mécanisme d'isolation. De plus, un mécanisme d'isolation logicielle est en général beaucoup plus flexible qu'un mécanisme matériel. On détaille ci-après l'implémentation d'un tel mécanisme dans l'exo-noyau de THINK.

Le contrôleur réseau

En plus d'implémenter un mécanisme de partage équitable de la bande passante, il paraît important de fournir un moyen de défense contre les attaques par inondation devenues courantes et face auxquelles de nombreux serveurs Internet se sont trouvés vulnérables. Aucun moyen de défense infaillible n'ayant été proposé jusqu'à présent, on propose des outils simples basés sur TCP/IP [13] et permettant de limiter le risque de saturation du système. Tout d'abord, le serveur peut filtrer les packets dont l'adresse IP source est visiblement forgée (par exemple une adresse en 192.168.0.0/16 pour un packet provenant d'Internet). Ensuite, une sentinelle peut être mise en place pour surveiller le nombre de connexions dans l'état SYN_RECEIVED et vider la file de connexion (*backlog queue*) avant qu'elle ne soit saturée. Cela n'empeche pas que des connexions légitimes échouent en cas de vidage de la file, mais cela évite un dépassement de capacité qui pourrait provoquer un arrêt du système. Enfin, le serveur peut considérer comme suspecte une adresse d'où sont parvenus beaucoup de packets SYN sans packet ACK correspondant et rejeter tous les nouveaux packets en provenance de cette adresse.

Le système de stockage persistant

Un partage équitable de l'espace de stockage nécessite la gestion de quotas d'espace disque par utilisateur, comme cela existe déjà dans la plupart des systèmes d'exploitation actuels. Cependant, cela n'offre aucune protection contre un programme malicieux qui effectuerait des lectures et ecritures répétées sur

le disque afin de ralentir les accès des autres applications. Beaucoup de travaux ont été effectués dans le domaine de l'ordonnancement sur disque (*disk scheduling*) [14][15], en basant le réordonnancement des requêtes sur des critères tels que l'ordre d'arrivée (*First Come, First Server*), la localisation du secteur sur le disque (*Shortest Seek Time First*), ou des contraintes temps-réel associées à la requête. Pour protéger le système contre des attaques visant à saturer le disque de requêtes, on peut mettre en place un tel méchanisme d'ordonnancement des requêtes, mais basés sur des critères de sécurités. Par exemple, un algorithme d'ordonnacement peut calculer le taux d'utilisation du disque par chaque processus, et faire passer en dernier les requêtes provenant d'un processus identifié comme monopolisateur.

4.2. Implémentation d'un mécanisme d'isolation logicielle

4.2.1. Présentation

L'algorithme utilisé est basé sur des techniques de validation de segment [16] et de greffage de code [17]. Son principe général est de générer dynamiquement (i.e. au moment où le code est chargé en mémoire à l'exécution) du code de vérification des accès mémoire. L'exemple ci-dessous en assembleur PowerPC illustre cette technique. Soit un processus exécutant le code ci-dessous, qui consiste à charger dans le registre r1 l'entier stocké à l'adresse r2 + 8 :

```
Code_initial:
lwz 1,8(2) // r1 := [r2 + 8]
```

Au moment où le processus est créé et son code chargé en mémoire, le mécanisme d'isolation parcourt ce code et génère le code de vérification associé à chaque instruction effectuant un accès mémoire. Ce code vérifie simplement que l'adresse pointée par l'accès mémoire est bien située dans l'intervalle autorisé. Cet intervalle est délimité par des bornes inférieures et supérieures stockées dans des registres (r15 et r16 sur le PowerPC). Puis il modifie le code initial pour remplacer l'accès mémoire par un branchement vers le code généré.

```
Code_modifie:
ba Code_genere // Branchement vers le code de vérification généré

Code_genere:
add 14,2,8          // r14 := adresse entier
tw 8,14,15          // Si adresse < borne inf alors exception
tw 16,14,16         // Si adresse > borne sup alors exception
lwz 1,0(14)         // r1 := [r14]
ba Code_initial + 4 // Retour au code initial
```

Cette technique permet de vérifier un code quelconque (on notera cependant qu'on tire parti du fait que les instructions du PowerPC sont toutes de la même taille (4 octets). Pour un processeur disposant d'un jeu d'instructions de tailles variables (ex. le Pentium d'Intel), il ne sera possible de mettre en oeuvre cette technique que si l'instruction d'accès mémoire est plus grande que le branchement qui doit la remplacer car on pourra alors compléter l'espace mémoire avec des instructions sans effet). Elle peut de plus être optimisée pour les instructions pour lesquelles l'adresse mémoire accédée est codée dans l'instruction elle même. C'est le cas par exemple de certains branchements pour le PowerPC. Dans ce cas, une simple vérification statique au moment du chargement du code est suffisante et on n'a pas besoin de générer de code.

4.2.2. Mesures de performances

Pour information, toutes les mesures effectuées l'ont été sur un PowerPC équipé d'un processeur G4 cadencé à 866 MHz et de 384 Mo de mémoire de type SDRAM PC100. Tous les programmes ont été compilés avec l'option -O de gcc, afin d'optimiser les accès aux variables locales.

Consommation mémoire :

Sur PowerPC, la taille moyenne du code généré pour un accès mémoire est 5 instructions. Evidemment, toutes les instructions d'un programme ne sont pas des accès mémoire, et nous avons donc calculé l'augmentation de l'espace mémoire occupé par le code sur un programme exemple. L'algorithme choisit est

un tri à bulles, qui effectue beaucoup d'accès à un tableau. Cet algorithme peut être codé en 29 instructions, donc 9 sont des accès mémoires. Parmis ces 9 accès mémoires, 4 sont des branchements relatifs dont la destination peut être vérifiée statiquement. On obtient donc au final un code généré de 25 instructions, soit presque autant que le code initial. Ce résultat peut paraître excessif, mais il faut garder à l'esprit que cette augmentation ne concerne que le code du programme. Or on sait que la taille du code d'une application est souvent très inférieure à celle de ses données, ce qui rend ce surcoût acceptable pour la plupart des machines, à l'exception des systèmes embarqués où la place mémoire est souvent très limitée.

Temps de génération :

Nous avons mesuré le temps nécessaire à la génération du code de verification pour un algorithme de 29000 instructions (i.e. l'algorithme de tri à bulles précédent copié/collé 1000 fois). Le délai mesuré est de 3.90 ms. Comme toutes les instructions sur PowerPC font 4 octets, on obtient un temps de traitement de 28 Mo/s. Sachant que la partie la plus coûteuse lors de la création d'un processus est la lecture du fichier ELF sur le disque lors de l'exécution de la fonction exec, et que le débit moyen d'un disque dur standard est de 15 Mo/s, on peut considérer ce délais comme acceptable et quasi-transparent pour l'utilisateur.

Coût à l'exécution :

Nous avons effectués des tests sur des programmes de types différents pour évaluer le délais à l'exécution engendré par l'algorithme d'isolation mémoire. Pour un branchement local, le surcoût est nul car la destination du branchement peut être vérifiée statiquement et aucun code n'est généré. Pour évaluer l'impact sur une lecture ou écriture mémoire isolée, nous avons mesuré le temps nécessaire à l'exécution de 2×10^9 lectures en mémoire d'un entier sur 32 bits, et obtenu un temps de 4621 ms sans vérification et 16175 ms avec, soit un temps d'exécution multiplié par 3,5. Pour l'algorithme de tri à bulles présenté ci-dessus, le temps d'exécution est multiplié par 2, puisque pour un tri de 100000 entiers, on obtient un temps de 56970 ms sans vérification et 116280 ms avec. Enfin, nous avons choisit un algorithme de calcul de racine carrée itératif (i.e. l'algorithme d'Héron d'Alexandrie), et obtenu un temps d'exécution de 6655 ms sans vérification et de 6758 ms avec, soit une augmentation de 1,55%, pour 10^6 calculs successifs de la racine carrée de 10^6. Ces deux algorithmes sont caractéristiques car le premier effectue beaucoup d'accès mémoire à l'interieur de deux boucles imbriquées, alors que le deuxième n'en effectue quasiment pas. On peut donc conclure que le surcoût engendré par la verification à l'exécution sera compris entre 0 et +100% selon la fréquence des accès mémoires.

4.3. Exemple d'utilisation

On détaille l'implémentation d'un ordonnanceur sécurisé utilisant les outils élémentaires présentés ci-dessus. Imaginons une application constituée de plusieurs processus. Le programmeur souhaite utiliser son propre ordonnanceur pour partager la ressource processeur entre ces processus, et il veut que ce partage soit équitable et qu'aucun processus ne puisse monopoliser le temps de calcul. Il utilise tout d'abord l'outil d'isolation mémoire logicielle afin de créer chaque processus dans son propre domaine de protection. L'ordonnanceur est lui aussi créé dans un domaine séparé. Pour pouvoir interrompre un processus en cours d'exécution et reprendre l'exécution d'un autre, il doit utiliser l'interruption gérant l'horloge système. Il enregistre donc un nouveau traitant d'interruption grâce à la fonction `TrapRegister` fournie par l'interface réifiant les interruptions. En faisant cela, il crée une liaison entre son ordonnanceur et le traitant d'interruption, grâce à l'usine à liaisons gérant les accès aux traitants d'interruptions. Pour s'assurer qu'aucun processus applicatif ne pourra faire de même et modifier le traitant d'interruption à son avantage, il suffit de surcharger la fonction `bind` de l'usine à liaison pour qu'elle vérifie que la requête provient bien d'un objet ordonnanceur autorisé et pas d'un processus applicatif. Pour vérifier que l'objet est bien autorisé à modifier le traitant d'interruption, l'usine à liaisons utilisera les outils d'identifications décris plus haut (par échange de capacités par exemple). On voit donc que le programmeur d'application peut facilement mettre en place un ordonnanceur sécurisé en utilisant simplement les outils élémentaires fournis par l'exo-noyau (ici : isolation logicielle, réification des interruptions), et les primitives du canevas logiciel (ici : usine à liaisons).

5. Conclusion

Comme on l'a vu, les architectures de noyau de systèmes d'exploitation actuelles ne sont pas satisfaisantes, et de nouveaux modèles sont nécessaires pour permettre au programmeur d'applications d'exploiter pleinement les ressources matérielles à sa disposition. L'architecture exo-noyau nous semble particulièrement bien adaptée à ces exigences de flexibilité et de performances, et l'architecture de systèmes répartis THINK intègre ces concepts dans un canevas logiciel à objet pour faciliter le développement d'application modulaires et adaptables. La prise en compte de la sécurité est bien entendu un point crucial lors du développement d'un système d'exploitation et on regrette que l'aspect préservation de la qualité de service soit souvent négligé dans les systèmes actuels. Pour permettre au concepteur de système de mettre en place sa propre politique de sécurité, on fournit dans THINK les outils élémentaires nécessaires à la mise en oeuvre d'une politique de sécurité quelconque. Ces outils sont identifiés et développés pour les quatre ressources matérielles principales d'un ordinateur, mais la même démarche peut bien sûr être appliquée à d'autres ressources. Les résultats préliminaires obtenus concernant l'implantation d'un outil élémentaire d'isolation mémoire logicielle semblent intéressants, bien qu'ils restent à les valider sur un système complet.

Bibliographie

1. Dennis Ritchie, Ken Thompson. The UNIX Time-Sharing System. Communications of the ACM, vol.17, num. 7, p. 365–375, 1974.
2. Dawson R. Engler, M. Frans Kaashoek. Exterminate All Operating System Abstractions. Workshop on Hot Topics in Operating Systems, 1995.
3. B. W. Lampson. On reliable and extendible operating systems. Proc. 2nd NATO Conference on Techniques in Software Engineering, Rome, 1969. Reprinted in The Fourth Generation, Infotech State of the Art Report 1, 1971, pp 421-444.
4. Michel Gien. Micro-Kernel Architecture, Key to Modern Operating Systems Design. Unix Review, vol. 8, num. 11, 1990.
5. Andrew S. Tanenbaum, Sape J. Mullender, Robbert van Renesse. Using Spare Capabilities in a Distributed System. Proceedings of the 6th International Conference on Distributed Computing Systems, 1986.
6. M. Rozier, V. Abrossimov, F. Armand, J. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Léonard, W. Neuhauser. Overview of the CHORUS Distributed Operating System. Chorus systèmes, 1990.
7. Robert V. Baron, David Black, William Bolosky, Jonathan Chew, Richard P. Draves, David B. Golub, Richard F. Rashid, Avadis Tevanian, Jr., Michael Wayne Young. Mach Kernel Interface Manual. Unpublished manuscript from the School of Computer Science, Carnegie Mellon University, 1990.
8. Przemyslaw Pardyak, Brian N. Bershad. Dynamic Binding for an Extensible System. Proceedings of the Second Symposium on Operating Systems Design and Implementation, 1996.
9. Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, Olin Shivers. The Flux OSKit: A Substrate for Kernel and Language Research. Proceedings og the 16th ACM Symposium on Operating Systems Principles, 1997.
10. Dawson R. Engler, M. Frans Kasshoek, James O'Toole Jr. Exokernel: An Operating System Architecture for Application-Level Resource Management. ACM Symposium on Operating Systems Principles, 1995.
11. M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Héctor M. Briceño, Russell Hunt, David Mazières, Thomas Pinckney, Robert Grimm, John Jannotti, Kenneth Mackenzie. Application Performance and Flexibility on Exokernel Systems. ACM Symposium on Operating Systems Principles, 1997.
12. Jean-Philippe Fassino and Jean-Bernard Stefani. Think : un noyau d'infrastructure répartie adaptable. 2ème Conférence Française sur les Systèmes d'Exploitation, 2001.
13. Information Sciences Institute for Defense Advanced Research Projects Agency. Internet Protocol, DARPA Internet Program Protocol Specification. RFC 791, 1981.
14. P. J. Denning. Effects of scheduling on file memory operations. AFIPS Spring Joint Computer Conference, 1967.
15. Bruce L. Worthington, Gregory R. Ganger and Yale N. Patt. Scheduling Algorithms for Modern Disk Drives. ACM Sigmetrics Conference, 1994.
16. Robert Wahbe, Steven Lucco and Thomas E. Anderson and Susan L. Graham. Efficient Software-Based Fault Isolation. ACM Special Interest Group on OPerating Systems, 1993.
17. Ariel Tamches, Barton P. Miller. Fine-Grained Dynamic Instrumentation of Commodity Operating System Kernels. USENIX, 1999.