Caractérisation de systèmes d'exploitation en présence de pilotes défaillants

Arnaud Albinet

Thèse de doctorat

Arnaud Albinet

CARACTÉRISATION DE SYSTÈMES D'EXPLOITATION EN PRÉSENCE DE PILOTES DÉFAILLANTS

Rapport LAAS n°

N° d'ordre : Année 2005

THESE

préparée au

Lab` rat` ire d'Analyse et d'Architecture des Systèmes (LAAS-CNRS)

en vue de l'obtention de

DOCTORAT DE L'INSTITUT NATIONAL POLYTECHNIQUE DE TOULOUSE

Spécialité : INFORMATIQUE ET TELECOMMUNICATIONS

par

Arnaud Albinet

CARACTÉRISATION DE SYSTÈMES D'EXPLOITATION EN PRÉSENCE DE PILOTES DÉFAILLANTS

Soutenue le 30 mars 2005 devant le Jury composé de :

Gilles Motet	Président
Gilles Muller	Rapporteur
Françoise Simonot-Lion	Rapporteur
Jean Arlat	Examinateur
Luc Bourgeois	Examinateur
Jean-Charles Fabre	Examinateur

Rapport LAAS n°

Cette thèse a été préparée au LAAS-CNRS, 7 avenue du Colonel Roche, 31077 Toulouse Cedex 04, France

Avant-propos

Les travaux présentés dans ce mémoire ont été effectués au sein du Laboratoire d'Analyse et d'Architecture des Systèmes du CNRS dont la direction a été assurée durant mon séjour par Messieurs Jean-Claude Laprie et Malik Ghallab, directeurs de recherche CNRS.

Je remercie Messieurs David Powell et Jean Arlat, directeurs de recherche CNRS, responsables successifs du groupe de recherche Tolérance aux fautes et Sûreté de Fonctionnement informatique (TSF) pour leur accueil et pour la confiance qu'ils m'ont accordée.

J'exprime ma reconnaissance à Messieurs Jean Arlat et Jean-Charles Fabre, qui ont dirigé mes travaux de thèse, pour m'avoir encadré et soutenu tout au long de cette thèse. Présents dans les moments « critiques », ils ont su mettre en œuvre les mécanismes de recouvrement par leurs conseils avisés. Leur soutien et leur confiance ont contribué largement au succès de cette thèse. Qu'ils trouvent ici le témoignage de mon estime et de ma reconnaissance..

Pour l'honneur qu'ils me font en participant à mon jury de thèse, je remercie :

- Madame Françoise Simonot-Lion, professeur à l'Institut National Polytechnique de Lorraine,
- Monsieur Jean Arlat, directeur de recherche CNRS,
- Monsieur Luc Bourgeois, chef de groupe logiciels embarqués chez Renault
- Monsieur Jean-Charles Fabre, professeur à l'Institut National Polytechnique de Toulouse,
- Monsieur Gilles Motet, professeur à l'Institut National des Sciences Appliquées de Toulouse,
- Monsieur Gilles Muller, professeur à l'Ecole des Mines de Nantes,

Ils ont contribué de manière significative à mes travaux : Tahar Jarboui et Marc Lecointe. Le premier a permis, par son expertise concernant la caractérisation des systèmes d'exploitation. Le second a participé à la réalisation technique et a mis les mains dans le cambouis avec moi. On peut partager seulement quand on est au moins deux !

Ces travaux ont été partiellement financés par la communauté européenne dans le cadre du projet DBench (*Dependability Benchmarking* IST-2000-25425).

Au début de mes travaux de thèse, j'ai également travaillé sur l'outil MAFALDA-RT (*Microkernel Assessment by Fault injection AnaLysis and Design Aid for Real Time systems*). Je remercie Manuel Rodríguez de m'avoir fait découvrir une partie des secrets de son outil.

Mes remerciements vont naturellement à tous les membres du groupe TSF, permanents, doctorants et stagiaires, ainsi que les membres des différents services techniques et administratifs du LAAS-CNRS, qui m'ont permis de travailler dans d'excellentes conditions.

Je remercie particulièrement Marco pour son soutien durant le doctorat et la rédaction de ce manuscrit. Je ne peux pas les nommer tous, mais ils ont contribué à faire de ce séjour au LAAS-CNRS une aventure agréable : les Nicos, le Ludo, Francois, Anna et Cristina, les frères Glu et Blu, Ali, Magnos et Roberta, Romain, ainsi que les joueurs de badminton et de foot.

La dernière pensée, mais non la moindre, reste pour Alex : merci à toi, pour tout.

S` mmaire

Intr` ducti` n générale

Les systèmes informatiques sont de plus en plus présents dans notre quotidien et les besoins en sûreté de fonctionnement ne se limitent plus aux domaines critiques classiques comme celui du ferroviaire, de l'avionique, du spatial ou du nucléaire. Les ordinateurs de bords embarqués dans les voitures, les serveurs sur Internet à rendement élevé (e-commerce, systèmes bancaires et transactionnels) constituent désormais des systèmes critiques par leurs impacts humain et financier. Une défaillance dans un de ces systèmes peut avoir des conséquences catastrophiques sur son environnement. En effet, imaginez les conséquences si le système ABS de votre voiture s'enclenche spontanément ou si votre banque en ligne est indisponible pendant trois jours. Longtemps, les systèmes critiques ont traditionnellement utilisé des composants matériels et logiciels spécifiques, élaborés par des concepteurs avertis des besoins de sûreté de fonctionnement du système global. Aujourd'hui, cette pratique est devenue souvent irréalisable à cause de son coût en temps et en main-d'œuvre. De plus, même dans ces conditions, l'élimination complète des fautes logicielles s'avère très difficile voire impossible compte tenu des contraintes de développement et de test. En effet, il est établi que la plupart des composants logiciels contiennent des fautes résiduelles lors de leur distribution.

Afin principalement de réduire les coûts de développement, la tendance est à l'utilisation de composants logiciels disponibles immédiatement ou « sur étagère ». Ils peuvent être commerciaux, souvent cités sous l'acronyme « COTS[1] ». En raison de l'opacité qui est souvent attachée à l'offre commerciale et les coûts significatifs pour obtenir le code source, l'option « source libre » se présente progressivement comme une alternative prometteuse. Les résultats de plusieurs études ont montré que les solutions « source libre » n'exhibaient pas nécessairement davantage de comportements critiques et ont même mis en évidence, dans certains cas, de meilleurs résultats que des solutions commerciales équivalentes. Dans ce manuscrit, nous nommerons COTS les composants disponibles sur étagères, commerciaux ou non.

Parmi ces composants, les logiciels exécutifs (comme les micronoyaux) sont des candidats privilégiés à l'intégration dans un système composé à partir de COTS. En effet, ils implémentent les services minimaux nécessaires à tout système informatique. L'intégration de COTS lors de la conception d'un système informatique s'étend désormais aux systèmes comportant des contraintes de sûreté de fonctionnement (y compris les systèmes critiques des domaines de l'aérospatial et de l'automobile, par exemple). Cependant, les concepteurs sont souvent peu disposés à entreprendre une telle démarche sans approfondir leur connaissance et leur compréhension du composant, particulièrement en ce qui concerne ses modes de

[1] *Commercial Off The Shelf*

défaillance et son comportement en présence de fautes. L'évaluation d'un COTS devient ainsi une étape importante lors de son intégration au sein d'un système soumis à des contraintes de sûreté de fonctionnement strictes.

Durant les dernières années, plusieurs études expérimentales ont abordé cette question selon différentes perspectives. Elles ont abouti à la proposition d'approches expérimentales d'étalonnage de sûreté de fonctionnement destinée à caractériser la robustesse des systèmes informatiques et des COTS. Cependant, ces propositions restent une étape préliminaire, et elles n'ont pas atteint le niveau de reconnaissance des étalons de performance. À ce titre, le projet DBench, auquel ce travail est associé et sur lequel nous reviendrons, vise à favoriser ces approches en définissant un cadre global pour la définition et la conduite d'étalons de sûreté de fonctionnement.

L'objectif de l'étalonnage de la sûreté de fonctionnement est de fournir un moyen générique pour la caractérisation du comportement des composants et des systèmes informatiques en présence de fautes par la quantification de mesures de sûreté de fonctionnement, dans l'objectif de les comparer. Au-delà des techniques, comme l'injection de fautes et les tests de robustesse, les étalons doivent fournir un moyen uniforme et efficace d'évaluation de la sûreté de fonctionnement. Dans la pratique, ces étalons doivent inclure :

- Une spécification précise des mesures de sûreté de fonctionnement à prendre en considération,
- Une spécification détaillée de toutes les procédures et étapes nécessaires pour obtenir ces mesures,
- Les domaines dans lesquels ces mesures sont valides et significatives.

La mise en place et la conduite d'expériences contrôlées forment la base de la caractérisation des systèmes informatiques. Pour comparer différents composants, cette caractérisation doit être capable de reproduire les observations effectuées sur un système et de généraliser les résultats à plusieurs systèmes. La notion de représentativité par rapport à l'étude en opération est considérée dans toutes les dimensions d'un étalon. Nous verrons cependant par la suite que, étant une simulation d'un domaine d'application, une analyse expérimentale d'un système d'exploitation reste, en dépit des informations objectives qu'elle procure, une représentation imparfaite de la réalité.

De récents travaux effectués sur le système d'exploitation *Windows* et une analyse du code source de *Linux* montrent qu'une proportion significative de défaillances d'un système d'exploitation provient de pilotes défectueux. Les pilotes de périphériques sont les logiciels qui permettent au système d'exploitation d'agir sur le matériel et les périphériques du système. Une grande partie du code d'un système d'exploitation se compose de ces programmes. Cette proportion augmente régulièrement : les versions récentes de *GNU/Linux* montrent qu'ils constituent désormais 60% du code. De tels pilotes sont généralement développés par des tiers, experts en matériel, et sont intégrés par la suite par les développeurs du noyau. Ce processus n'est pas toujours maîtrisé et un comportement incorrect de ces programmes, intimement reliés au noyau, peut avoir des effets néfastes sur le système global.

Pour résoudre ce problème, il est nécessaire d'étudier et de proposer des méthodes pour analyser spécifiquement l'impact des pilotes défectueux sur les logiciels exécutifs. La collecte

des données relevées en fonctionnement ne permet pas l'analyse des comportements spécifiques du composant, la fréquence d'activation des fautes résiduelles logicielles étant trop faible. Elle demandait de vastes échantillons de composants étudiés ainsi qu'une période d'étude souvent incompatible avec les contraintes économiques du domaine.

À notre connaissance, peu de recherches ont porté sur ce sujet. Au niveau des pilotes de périphériques, les travaux se concentrent plutôt sur le problème de la fiabilité du noyau ou de la caractérisation de la robustesse des logiciels de pilotage soumis aux défaillances du matériel. Cependant, comme nous verrons par la suite, certains travaux sur les systèmes *Windows* et *GNU/Linux* traitent de l'impact des fautes présentes dans les pilotes sur le système. Le travail rapporté dans ce manuscrit complète ces recherches en proposant une approche alternative plus ciblée.

Afin de caractériser le comportement du système en présence de fautes, nous avons besoin de simuler leur activation. Les techniques d'injection de fautes, où des comportements défectueux sont délibérément provoqués pour simuler l'activation des fautes, fournissent une approche pragmatique et adaptée pour soutenir l'analyse spécifique d'un logiciel. Elles font d'ailleurs partie du processus de validation chez plusieurs éditeurs de logiciels. Parmi ces techniques, celle d'injection de fautes par logiciel fournit le niveau approprié de flexibilité et d'intrusivité. Sur la base de ces principes, nous avons développé une méthode pour l'évaluation de la robustesse des noyaux face aux comportements anormaux des pilotes. Nous avons également défini différentes techniques d'analyse et d'interprétation des résultats obtenus par cette méthodologie. Ces interprétations prennent en compte différents points de vue afin de répondre au besoin réel de sûreté de fonctionnement de l'utilisateur de la technique. Nous avons enfin illustré l'applicabilité de cette méthode par sa mise en œuvre dans le cadre d'un environnement expérimental sous Linux.

Après avoir analysé et précisé les caractéristiques de l'interface pilote-noyau (Driver Programming Interface - DPI), nous avons proposé une méthode originale d'injection de fautes basée sur la corruption des paramètres des fonctions manipulées au niveau de cette DPI. De cette façon, la définition de l'injection de fautes exprime plus précisement l'impact des divers types d'interactions entre les pilotes et le noyau. En outre, les erreurs provoquées peuvent simuler efficacement les conséquences de fautes de conception pouvant affecter les programmes, et en particulier les pilotes. La caractérisation du système d'exploitation en présence de pilotes défaillants permet ainsi d'obtenir des informations objectives pour :

- Identifier les sections les plus fragiles de l'interface pour mettre en place des mécanismes de prévention et de tolérance aux fautes afin d'améliorer la sûreté de fonctionnement du système.
- Comparer le comportement de différents systèmes d'exploitation vis-à-vis d'erreurs provenant des pilotes de périphériques grâce à une approche portable et reconnue.

***

Ce mémoire est structuré en quatre chapitres :

Le premier chapitre motive nos travaux sur les pilotes de périphérique. Nous identifions et analysons tout d'abord les causes possibles des défaillances dans les pilotes de périphériques. Nous décrivons ensuite la structure et le rôle d'un pilote au sein d'un système informatique et présentons les similarités des différentes architecture des pilotes de périphériques des systèmes étudiés, comme *Windows XP*, *GNU/Linux* et *MacOSX*. Le chapitre fournit également une présentation rapide des principales techniques d'amélioration de la sûreté de fonctionnement des systèmes d'exploitation : plusieurs mécanismes de prévention et de tolérance aux fautes ont déjà été mis en œuvre concernant les pilotes pour réduire ou éviter les effets de leurs défaillances.

Le deuxième chapitre présente le cadre conceptuel de l'étalonnage de systèmes d'exploitation dans lequel nous avons cherché à inscrire notre démarche. Nous nous attachons notamment à préciser les contraintes de conception et de réalisation d'une méthode d'évaluation de la sûreté de fonctionnement. Plusieurs techniques ont déjà été mises en application sur différents COTS. Nous décrivons également dans ce chapitre les outils correspondants qui ont servi de support de réflexion à l'élaboration de notre approche. Enfin, nous concluons en précisant la problématique abordée par notre travail : la prise en compte de l'influence des défaillances des pilotes de périphériques sur la sûreté de fonctionnement du système.

Le troisième chapitre présente la méthode que nous proposons pour l'évaluation du comportement des systèmes d'exploitation en présence de pilotes défaillants. La définition de l'interface entre les pilotes et les noyaux en est le point central. Le choix de la solution par injection de fautes pour l'évaluation est justifié par ses nombreux avantages. Nous précisons l'étape de construction des modèles de fautes et quels sont leurs objectifs. Les différents principes pour la définition de la charge de travail sont également précisés. Ensuite, nous montrons comment obtenir des mesures pertinentes dans le cadre d'une analyse expérimentale et les types de résultats que nous pouvons obtenir. Enfin, nous examinerons différentes options pour l'analyse des résultats selon le contexte de sûreté de fonctionnement et le point de vue de l'utilisateur.

Le quatrième chapitre met en œuvre la méthode en l'appliquant à un cas d'étude : Le système d'exploitation « source libre » GNU/*Linux*. Nous décrivons tout d'abord la plate-forme expérimentale qui permet d'évaluer les problèmes liés aux pilotes de périphériques. Le noyau *Linux* a été choisi pour sa notoriété dans le milieu informatique et la disponibilité des informations le concernant. De plus, le choix d'un noyau « source libre » comme première cible a été un facteur important pour la mise au point de notre méthode. La conception de la plateforme doit tenir compte des mesures recherchées et des contraintes physiques liées à de telles expériences. La conception de la charge de fautes et de l'activité est détaillée. Nous présentons et commentons les analyses possibles à partir des résultats obtenus par l'étude expérimentale. Nous montrons que la comparaison des résultats selon divers points de vue peut conduire à des conclusions différentes. Enfin, nous terminons le chapitre en montrant comment une analyse détaillée des comportements de certaines fonctions permet de mieux comprendre l'impact des pilotes sur le noyau *Linux*.

Chapitre 1 La pr` blématique des pil` tes

Aujourd'hui, les logiciels de gestion de périphérique posent un problème majeur en termes de sûreté de fonctionnement. De nombreuses études réalisées sur différentes plates-formes montrent que les pilotes sont fréquemment à l'origine des dysfonctionnements des systèmes d'exploitation. Ce phénomène n'épargne aucun des systèmes d'exploitation répandus comme ceux des familles *Windows* et GNU/*Linux*.

Les raisons du taux élevé des défaillances dans les pilotes sont multiples et sont abordées dans ce chapitre. La conception et la réalisation d'un pilote s'effectuent dans des conditions difficiles, et la réalisation d'un pilote de périphériques performant et sûr de fonctionnement nécessite des connaissances approfondies de la part du concepteur et du développeur. L'environnement particulièrement complexe d'un logiciel de gestion de périphérique amplifie ces difficultés.

Le succès des nouveaux périphériques informatiques aggrave l'impact des pilotes. En effet, pour répondre à l'augmentation croissante et rapide de matériels gérés par le système (caméras, cartes diverses), les systèmes d'exploitation commerciaux sont étendus en chargeant du code peu sûr directement lié avec le noyau. Nous présentons dans ce chapitre les tentatives pour diminuer les risques dans le noyau. Elles exigent souvent de changer la manière d'écrire le code ou d'effectuer des modifications dans la structure des systèmes d'exploitation. De telles approches sont donc parfois difficiles à mettre en œuvre dans le cas des modules de gestion de périphérique, qui sont les extensions les plus communes des systèmes d'exploitation et représentent un investissement énorme en temps d'élaboration. Par conséquent, ces approches n'intègrent pas, à ce jour, les systèmes d'exploitation traditionnels.

Aujourd'hui, le besoin des utilisateurs de systèmes n'est pas simplement l'analyse ou la détection de la défaillance, mais également le rétablissement rapide du système pour pouvoir bénéficier de ses services. Par conséquent, le système d'exploitation ne doit pas simplement isoler les modules de gestion de périphérique défectueux, mais leur permettre également de reprendre rapidement le service, après un redémarrage ou après une procédure de recouvrement du système en exploitation (sans redémarrage de la machine). À l'avenir, il est clair que l'amélioration de la sûreté de fonctionnement du système d'exploitation dépendra de la sûreté de fonctionnement des pilotes car le noyau n'est plus la source principale des bogues. Dès lors que le logiciel mûrit et que les dispositifs d'intégration se rétrécissent, les défaillances de matériel (et les fautes physiques) deviendront un plus grand problème. En conséquence, les logiciels d'exploitation doivent permettre de :

- Tolérer et récupérer des pilotes défectueux,
- Tolérer et récupérer du matériel défectueux.

Dans ce chapitre, nous justifions dans un premier temps l'intérêt porté aux problèmes liés aux pilotes de périphériques dans l'étude de la sûreté de fonctionnement des systèmes d'exploitation. Ensuite, nous identifions les origines physiques et logiques possibles de ces problèmes. Après une brève présentation des systèmes d'exploitation, nous explicitons le rôle du pilote et les différentes architectures des pilotes des systèmes d'exploitation les plus connus. Ensuite, nous motivons les techniques de sûreté de fonctionnement mises en œuvre pour pallier les problèmes présentés par les pilotes. En effet, différents travaux pour tolérer et éviter les fautes provenant des extensions du noyau ont déjà été effectués. La nécessité d'évaluer un système, en prenant en compte ses efforts vis-à-vis de la sûreté de fonctionnement du point de vue de ses extensions, est une des principales motivations de nos travaux.

1.1 S` urce imp` rtante d'erreurs

Nous allons voir dans ce paragraphe les différentes recherches effectuées durant les dix dernières années sur les origines des dysfonctionnements des systèmes d'exploitation. En 1999, la principale cause d'instabilité du système d'exploitation *Windows NT* [Russinovich 99] était attribuable aux modules de gestion de périphériques. Lorsque le système fut largement adopté à travers le monde, le nombre associé d'applications augmenta considérablement. En outre, la plupart des périphériques fonctionnant sur une architecture x86 étaient disponibles pour *Windows NT*. Son éditeur, la firme *Microsoft*, estime qu'environ 12000 applications et 24000 périphériques sont associés à NT. Bien que la mise à disposition d'un grand nombre d'applications et de périphériques fasse de NT un système attrayant, facile à employer et à mettre en œuvre, cette richesse augmente aussi ses faiblesses. Les modules de gestion de périphérique mal écrits donnent aux utilisateurs l'impression que NT est un système instable. Même si les pilotes certifiés HLC (certification établie par *Microsoft*) sont légèrement moins sujets à défaillances, ils restent une importante source d'erreurs. *Microsoft* n'avait fourni jusque-là que des outils limités aux administrateurs. Le fardeau du développement correct des applications et des pilotes revenait à leurs concepteurs, pas toujours aptes à relever le défi en termes de sûreté de fonctionnement.

Mais le problème ne se réduit pas aux systèmes d'exploitation de la firme de Seattle. En 2001, une étude conduite à l'université de Stamford a révélé que les pilotes *Linux* (pour une architecture x86) représentent soixante-dix pour cent des lignes de code du noyau [Chou *et al.* 2001]. Durant les dix dernières années, l'augmentation par dix de la taille du noyau est presque exclusivement due à l'augmentation du nombre de pilotes. Dans le même temps, le reste du noyau conserve une taille stable. Ce n'est donc pas une surprise de voir une grande partie des erreurs affectant le code du système être localisée au niveau des pilotes. Représentant la plus forte part en termes de lignes de code, ils risquent de contenir plus d'erreurs de programmation que le reste du système. Mais les résultats obtenus par une analyse statique dans [Chou *et al.* 2001] avec un ensemble de vérifieurs révèlent un autre constat : La fréquence des bogues par lignes de code dans les pilotes est trois à sept fois supérieure à celle du reste du système d'exploitation. Plus récement, la société Coverity a annoncé dans [LWN Coverity 2004] que son produit SWAT (un analyseur statique de code très sophistiqué mais sous licence propriétaire) avait décelé 985 bogues sur un total de 5.7 millions de lignes de code dans le noyau Linux 2.6.9. La majorité des bogues se trouve dans les modules des pilotes et seulement 1% dans le noyau. Le ratio est donc d'environ 0.17 bogues par millier de lignes ce qui, selon l'université Carnegie Mellon, est beaucoup moins que les logiciels classiques.

Comme présentés sur la Figure 1-1, les pilotes composent la plus grosse partie des lignes de code de *Linux*, mais sont également la partie de code la plus dense en termes d'erreurs. La combinaison de ces deux facteurs aggrave considérablement l'influence des pilotes sur la sûreté de fonctionnement des systèmes.

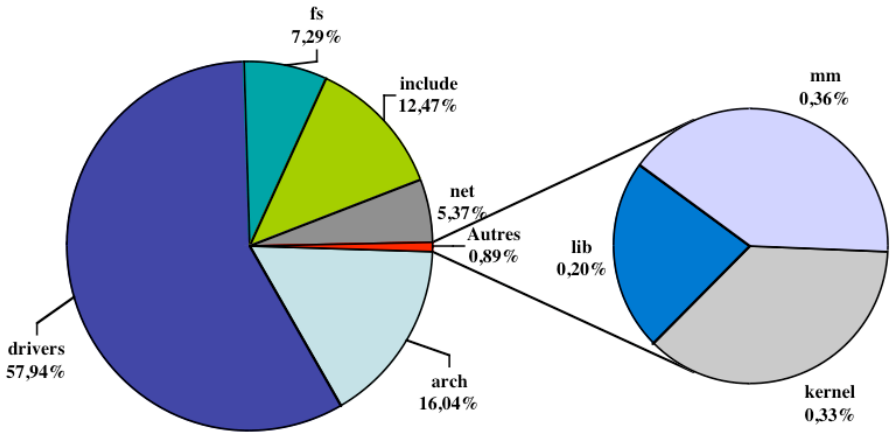


Figure 1-1 Répartition des lignes de codes dans le noyau Linux 2.6

En intégrant des concepts de sûreté de fonctionnement lors des perfectionnements de *Windows 2000*, *Microsoft* a donné aux développeurs des outils pour corriger les problèmes dans les applications et les pilotes pendant leur conception, lors de la phase de test. Cependant une analyse en opération des défaillances de *Windows 2000* a montré que les modules de gestion de périphériques représentent encore 27% des incidents, comparé à 2% pour le noyau lui-même [Murphy & Levidow 2000]. Ce problème est aggravé par la popularité de nouveaux périphériques, comme les appareils numériques pour le grand public (caméras par exemple), qui augmentent le nombre de pilotes à fournir et donc le nombre de défaillances qui leur sont associées.

Les efforts fournis dans *Windows XP* se composent, entre autres, d'un environnement dédié au développement de pilotes (*Device Developement Kit*) ayant pour objectif de consolider la programmation des pilotes par l'utilisation d'une interface accessible et claire. Malgré cela, en 2003, les pilotes causent encore 85% des échecs rapportés dans *Windows XP* selon un article interne rédigé par Rob Short[2].

La fréquence des erreurs de programmation reste largement supérieure dans les modules de gestion de périphérique que dans le reste du noyau. Le phénomène est aggravé par le fait que cette fréquence d'erreurs élevée concerne la plus grosse partie du système en opération. Alors que le noyau du système atteint des niveaux élevés de fiabilité grâce à sa longévité et aux essais répétés, le système étendu (comprenant les pilotes) reste encore la source de nombreuses défaillances. Nous allons voir dans le paragraphe suivant leurs origines.

---

[2] Rob Short : Vice président de la division *Windows Core Technology* de *Microsoft*

1.2 Origines des fautes liées aux pil` tes ` u aux matériels

Nous allons dans cette section établir les principales causes de la présence de fautes dans les pilotes et des extensions en général. D'après la terminologie présentée dans [Laprie *et al.* 1995] sur les liens entre faute, erreur et défaillance, l'origine des fautes est l'élément ayant engendré un comportement anormal que l'on souhaite prévenir ou tolérer. Cet élément peut être de nature physique (matériel ou humain) ou logique (logiciel).

1.2.1 Les fautes logicielles

Les fautes de nature logicielle résultent généralement de compromis effectués durant le développement du programme dans l'objectif de conserver le système à un niveau de performance acceptable, afin de faciliter sa conception, ou encore pour des raisons économiques. Elles peuvent aussi intervenir lors d'interactions non prises en compte pendant la phase de conception et qui se révèlent seulement lors de la mise en opération. La complexité intrinsèque de la conception d'un pilote de périphérique est importante : le pilote doit allier performance, sûreté de fonctionnement et compatibilité avec les différentes versions de matériels et de systèmes.

Les spécifications informelles lors de la conception peuvent être aussi une importante source de difficulté. Les travaux rapportés dans [Gaudel 1991] expliquent comment les spécifications sont une source potentielle de malentendu lors de la conception d'un logiciel. En effet, le langage humain est souvent soumis à l'interprétation, d'une part lors de la phase de conception du cahier des charges et d'autre part lors de la phase d'implémentation. La formalisation des besoins du « client » (l'entité souhaitant le produit) peut conduire le concepteur à l'interprétation. D'une manière similaire, l'intégrateur ou le développeur doit interpréter le cahier des charges pour réaliser le logiciel demandé. Ces interprétations sont fréquemment la cause d'erreurs lors de la conception d'un logiciel. La terminologie propre au domaine des pilotes de périphériques (programmation noyau) et le mélange des niveaux d'abstraction accentuent les limites du langage humain. Les risques d'interprétations biaisées augmentent et l'utilisation de moyens traditionnels de conception de logiciels montre ses limites. L'utilisation d'un langage spécifique au domaine permet de résoudre ce problème et sera présentée à la fin de ce chapitre.

Il convient de rappeler que les pilotes sont souvent écrits par des fabricants de matériels (contrôleurs ou périphériques) plutôt que par les développeurs de systèmes d'exploitation. Ils ne disposent donc pas d'une expérience étendue de la programmation dans le noyau comme ces derniers. La compréhension globale de la gestion des périphériques par un noyau peut s'avérer cruciale lors de l'implémentation d'un pilote. Dans certains cas, les fabricants de matériels développant des pilotes peuvent ne disposer que de peu de connaissances à ce sujet. Par ailleurs, les documentations de la programmation noyau sont destinées à un public averti, souvent difficile d'accès et parfois même confuses. Les connaissances nécessaires à la bonne réalisation d'un pilote sont reparties sur les trois couches informatiques :

- la couche applicative, utilisatrice du système;
- la couche système hébergeant le noyau;
- la couche matérielle impliquée par le pilote que l'on désire concevoir.

Les développeurs d'un tel logiciel doivent connaître de manière approfondie l'ensemble de ces couches afin d'anticiper les problèmes d'interactions auxquels ils peuvent être confrontés. Or, cette compétence ne se trouve pas fréquemment dans les sociétés fournissant des dispositifs matériels, mais plutôt chez celles qui produisent les systèmes d'exploitation. Elle manque souvent dans les « petites » entreprises souhaitant fournir les pilotes de leurs périphériques.

Concernant la couche système, la programmation de pilotes exige souvent de comprendre le fonctionnement de dispositifs asynchrones complexes et également de protocoles de commande évolués. Le nombre de personnes ayant des connaissances approfondies sur les techniques propres aux systèmes d'exploitation n'évolue pas aussi rapidement que la quantité de pilotes à fournir. Il est donc de plus en plus difficile de trouver du personnel qualifié pour ce genre de développement. Toutefois, nous verrons à la fin de ce chapitre les solutions envisagées pour réduire cette difficulté.

Une des conséquences découlant de la difficulté à concevoir et développer de tels logiciels est que les pilotes sont fréquemment élaborés en copiant et en éditant le code des pilotes existants, souvent sans maîtrise complète des fonctionnalités réutilisées. Dans des soucis d'économie d'argent et de temps, ce phénomène est très fréquent dans de nombreuses équipes de développement, en particulier chez les fournisseurs de matériel, qui représentent une part importante des concepteurs de pilotes. Facilitant la programmation dans un premier temps, ce genre de pratique conduit à des bogues subtiles, difficilement identifiables comme celui du masquage d'interruption. En effet, la présence des pilotes au cœur du noyau peut amener des problèmes difficilement analysables par le développeur, car les bogues se révèlent généralement en opération et leur correction est jugée trop coûteuse pour être mise en œuvre.

L'environnement de la programmation dans le noyau reste très particulier. Il est en général faiblement documenté, que ce soit pour la gestion mémoire, la gestion des accès concurrents ou les règles de préemption. Les outils de développement d'un pilote ne sont pas aussi pratiques que ceux d'une application classique. Les débogueurs sont souvent inefficaces et, de manière plus générale, les IDE (*Integrated Development Environment*) complets n'ont fait leur apparition que dernièrement. Les contraintes de développement sont plus élevées et les possibilités d'évaluer et de tester les pilotes sont donc réduites. Les bogues ont souvent un fort impact sur le système ce qui rend leur compréhension et leur correction très difficiles. La programmation dans le noyau est faiblement protégée, laissant une grande liberté aux programmeurs dans l'objectif de réaliser des performances. Cette liberté est souvent la source de faiblesses du système. En effet, une part importante de la performance du système provient des logiciels de pilotages. Certains périphériques sont régulièrement sollicités lorsque le système est en opération. Alourdir ces programmes en effectuant des contrôles lors de leur exécution conduit à une réduction sensible de la performance. Pour Linus Torvalds[3], la baisse de performance n'est pas admissible dans le noyau d'un système d'exploitation. Toutefois, avec les progrès matériels modernes, les puissances de calculs sont suffisamment importantes pour rendre une perte de performance négligeable par rapport au gain de sûreté de fonctionnement, en particulier dans des systèmes critiques. Dans ce cadre, la mise en place de systèmes de recouvrement (*wrapper*) a permis des progrès en termes de sureté de

[3] Instigateur de Linux

fonctionnement comme sur le module de synchronisation du micronoyau de première génération *Chorus* dans [Rodriguez *et al*. 2002].

Les techniques actuelles de conception de pilotes de périphériques comportent de grosses lacunes en terme de qualité de conception logicielle. Les conditions de développement sont souvent très difficiles. Les fautes logicielles sont la principale cause d'erreurs liées aux pilotes. Toutefois, les fautes matérielles représentent une deuxième source d'erreurs. Aujourd'hui faible, sa part risquerait d'augmenter avec la venue d'amélioration de la prévention de fautes logicielles.

1.2.2 Les fautes physiques

Les fautes physiques sont dues à des phénomènes physiques adverses. Elles peuvent être:

- Internes: elles proviennent d'une partie défaillante du système. L'activation de cette partie produisant une erreur;
- Externes: elles résultent de l'interférence ou des interactions du système avec son environnement physique.

Les avancées permanentes de la technologie électronique permettent d'intégrer de plus en plus de transistors sur un seul circuit (VSLI et ULSI), de plus en plus de fonctionnalités sur une seule carte, voire sur un même circuit (SOC). Une analyse de l'évolution des circuits microprocesseurs et mémoires durant les trente dernières années montre un accroissement du nombre de transistors par an de 40% et de 60% pour les mémoires. On peut intégrer aujourd'hui plus d'un milliard de transistors sur un seul circuit. Des mémoires DRAM de plusieurs gigabits existent avec des règles de dessins de 0,1 micromètres. Moins d'un dixième de millimètre sépare chaque composant afin d'accroître la rapidité de traitement de l'information et son transfert. De plus, la loi de Moore énonce que la capacité de transfert des bus double tous les 18 mois. On peut alors constater que les travaux des dernières années ont été portés sur la performance plus que la qualité et la sûreté de fonctionnement. D'après de récents rapports de la société *Intel* [Edwards & Matassa 2002], il devient de plus en plus difficile aujourd'hui de produire des composants exempts de fautes de conception ou de production.

Les perturbations électromagnétiques, les radiations, la température, les vibrations, un matériel défectueux peuvent être à l'origine de fautes physiques. Ces fautes peuvent avoir un impact sur n'importe quel système. Aujourd'hui, les fautes physiques à l'origine d'une défaillance des systèmes d'exploitation les plus répandus représentent une part négligeable des fautes. Ce raisonnement ne serait pas valable dans le cas de domaine très particulier comme, par exemple, le domaine spatial. Nous verrons que certaines techniques existent pour réduire l'impact de ce type de fautes à la fin de ce chapitre. Les fautes logicielles restent donc la principale origine des fautes liées aux pilotes. Pour mieux cerner leurs effets possibles, nous présentons dans le prochain chapitre la composition sémantique puis logicielle d'un pilote de périphériques.

1.3 Les pil`tes de périphériques dans les systèmes d'expl`itati`n

Certains pilotes de périphériques sont essentiels pour le système, comme les pilotes de bus. Ils font généralement partie intégrante du noyau. Toutefois, la plupart des logiciels de gestion de périphériques se présentent sous la forme d'extensions du noyau, rajoutée après la mise en service du système. Ces extensions fournissent des services qui ne sont pas gérés par le cœur du noyau. Nous allons voir dans ce paragraphe quel rôle tient un pilote dans un système d'exploitation moderne.

1.3.1 Rôle

Les systèmes d'exploitation modernes sont constitués de plusieurs composants comme le gestionnaire de mémoire, le gestionnaire d'ordonnancement des processus ou encore la couche d'abstraction matérielle (HAL). Le noyau du système d'exploitation peut être vu comme une « boîte noire » qui apporte un support aux applications pour leur exécution, via des méthodes pour interagir avec le matériel existant mais aussi le matériel à venir.

Cependant, lors de sa distribution, un noyau qui inclurait toutes les fonctionnalités pour interagir avec le matériel existant serait peu pratique, car il utiliserait beaucoup de ressources dont certaines inutilement. De plus, le système ne pourrait gérer les nouveaux matériels, ceux qui n'existaient pas encore lors de sa distribution. En effet, la fréquence de commercialisation de nouveaux matériels informatiques augmente régulièrement. Ce phénomène atteint son apogée avec les cartes graphiques : aujourd'hui, la société *NVidia*, spécialisée dans la conception de cartes graphiques, fournit une nouvelle carte tous les 6 mois et un nouveau pilote tous les 3 mois. Un système d'exploitation est amené à s'interfacer avec du matériel qui n'existe pas lors de son élaboration ou de sa distribution. Dans cet objectif, les noyaux des systèmes modernes peuvent étendre leurs fonctionnalités par l'ajout en opération d'extensions permettant la gestion de ces nouveaux périphériques. Ceci motive la perception du pilote comme un élément externe au noyau du système d'exploitation.

Chaque extension implémente plusieurs routines pour interagir avec le nouveau composant. Les extensions concernent autant les pilotes de périphériques matériels que les pilotes d'abstractions logicielles. La plupart des utilisateurs achètent un noyau avec des extensions « classiques », puis adaptent le système à leurs besoins grâce à des extensions et des applications.

Le rôle du pilote de périphérique dans le système est de fournir une interface abstraite du périphérique pour le système d'exploitation comme présenté dans la Figure 1-2. Dans le système d'exploitation, les pilotes sont la partie contenant les instructions qui dépendent directement du matériel. Les composants situés au-dessus (le système d'exploitation et les applications) dans la Figure 1-2 peuvent utiliser de manière transparente le matériel. Le pilote implémente un ensemble de fonctions (lire, écrire, agir, ...) qui activera le périphérique à travers son contrôleur. Le contrôleur présent sur le périphérique est la partie du matériel qui fournit une interface matérielle « haut niveau » avec le périphérique. Les pilotes utilisent cette interface pour dialoguer avec le périphérique et satisfaire les demandes de la partie logicielle du système. À partir des pilotes, les instructions ne dépendent plus de l'architecture de la machine. Les pilotes définissent quand et comment les périphériques interagissent avec le

reste du système. Ils forment ensemble le gestionnaire de périphériques du système d'exploitation.

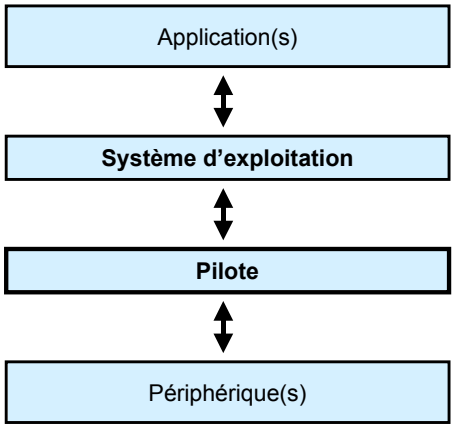


Figure 1-2 Rôle du pilote dans un système

1.3.2 Organisation fonctionnelle

Nous présentons dans ce paragraphe les raisons de la position particulière des pilotes dans le système. Le contenu des pilotes est organisé en trois parties distinctes (voir Figure 1-3) nécessitant chacune l'accès à des instructions privilégiées du système : La partie gérant la politique d'échanges avec le noyau, la partie code permettant la réalisation des fonctionnalités disponibles sur le périphérique et enfin la partie dépendante du matériel. Pour cela, et pour

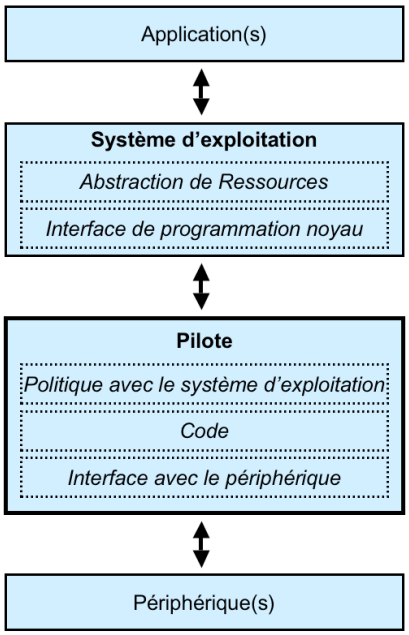


Figure 1-3 Contenu d'un pilote

indiquer les nouvelles fonctionnalités mis à sa disposition, les pilotes interagissent fréquemment avec le noyau.

Dans la partie « politique » avec le système d'exploitation, les pilotes définissent quand et comment le monde matériel et le monde logiciel interagiront. Elle contient la gestion globale du périphérique et du pilote en définissant ses points d'entrée disponibles. Par exemple, dans le cas d'un pilote de type scrutation, une application effectue une demande par l'intermédiaire d'un appel au noyau (ouverture, lecture, ...) afin d'accéder à des données sur un périphérique (une clé USB, par exemple). Le processeur passe en mode superviseur et exécute le code du pilote correspondant à l'opération appropriée. Une fois l'opération finie, le pilote libère le processeur qui reprend alors l'exécution de l'application en mode d'utilisateur.

La partie « code » du pilote, représentée sur la Figure 1-3, implémente un ensemble de fonctions de base qui agiront sur le périphérique. Les instructions correspondantes dépendent du noyau du système et du type de périphérique piloté. Un module de gestion de périphérique implémente généralement les fonctionnalités suivantes :

- Enregistrement et désenregistrement du pilote : permettent d'insérer et supprimer le code du pilote du système d'exploitation.
- Insertion et suppression du périphérique dans le système : initialise le périphérique lors de son ajout au système et purge le système lors de son retrait.
- Ouverture et fermeture d'un périphérique,
- Opérations de contrôle sur un périphérique (écriture, lecture, entrées/sorties),
- Traitements des interruptions et des signaux,
- Diverses options possibles (« *Plug and Play* », alimentation,...).

La partie « interface » avec le périphérique met en œuvre les mécanismes directement liés à l'architecture matérielle du périphérique, souvent via d'autres pilotes de périphériques de niveau inférieur (par exemple, un pilote de bus). Elle est souvent écrite dans des langages bas niveau (C voire assembleur) et permet d'accéder à la partie matérielle du périphérique via les ports et les registres du périphérique. Nous aborderons plus précisément la composition des logiciels de pilotage de périphériques au cours de ce chapitre.

1.3.3 Commentaires

Dans les systèmes d'exploitation actuels, les pilotes de périphériques ont une situation très particulière. Ils sont à l'intérieur du système d'exploitation tout en étant développés par des personnes externes à la conception du noyau de ce système. Leur position fragilise le système d'un point de vue conception de logiciel. Les interactions avec le noyau représentent alors des éléments essentiels de la protection du noyau vis-à-vis de comportement anormal de pilotes. La combinaison de privilèges d'exécution élevés avec un espace d'adressage partagé avec le noyau permettent de réaliser n'importe quelle opération dans la plupart des systèmes. Ces différents points ont été malheureusement nécessaires pour le maintien des performances du système. En effet, les pilotes sont souvent fortement sollicités au cours de l'exploitation du système d'exploitation. Une baisse de performance d'une simple fonctionnalité d'un pilote peut avoir un impact significatif sur les performances du système compte tenu de l'utilisation

fréquente de cette fonctionnalité. Nous présenterons à la fin de chapitre les travaux effectués sur différents systèmes pour prévenir et tolérer les fautes provenant d'un pilote de périphérique.

1.4 Les systèmes d'expl` itati` n

Nous fixerons tout d'abord les caractéristiques principales des systèmes d'exploitation, ou « systèmes d'exploitation » dans le langage commun, nécessaires à la comparaison de leurs architectures. Nous présenterons ensuite les similarités et les différences des architectures les plus répandues afin de faciliter la définition d'un prototype d'étalon de sûreté de fonctionnement générique pour les systèmes d'exploitation. En effet, un étalon est avant tout un standard portable, son objectif est de pouvoir comparer les résultats obtenus avec différentes solutions (par exemple différents systèmes d'exploitation).

1.4.1 Architectures et concepts

Selon [Nutt 2000], les systèmes d'exploitation ont un rôle consistant à :

- Fournir les services de base aux applications à travers l'interface de programmation d'applications (API pour *Application Programming Interface*).
- Partager les ressources matérielles (mémoire, disque, etc.) et logicielles (files d'attentes, sémaphores, etc.) entre les applications,
- Fournir une abstraction du matériel sous-jacent (HAL pour *Hardware Abstraction Level*).

Il existe différentes variétés de systèmes d'exploitation et chaque domaine d'utilisation en dispose d'un ou plusieurs dédiés. Ils ont par ailleurs suivi l'évolution du monde informatique : ils peuvent être commerciaux ou à code source libre. Les systèmes d'exploitation sont conçus pour une classe spécifique d'utilisateurs ou d'applications et ils sont généralement dédiés à une architecture matérielle spécifique. Il est donc difficile d'examiner un système sans l'architecture qui lui est associée.

Le modèle de programmation pour la plupart des noyaux s'appuie sur une abstraction fondamentale : les processus. Un processus peut être en exécution, en pause ou terminé. On appelle espace d'adressage la partie de la mémoire virtuelle dans laquelle le processus effectue ses opérations. On appelle privilèges d'exécution les droits que l'on accorde à un processus. Plus les droits sont élevés, plus le nombre d'instructions autorisées augmente.

Les privilèges d'exécuti` n

Un système d'exploitation est divisé en plusieurs niveaux de privilèges. La partie du système d'exploitation qui nécessite l'accès au matériel et qui fournit les abstractions de base, comme les processus et la mémoire, s'exécute dans le niveau de plus haut privilège. Quelques tâches systèmes (comme les commandes de supervision) s'exécutent dans un niveau de moindre privilège. Enfin, les processus utilisateurs s'exécutent dans le niveau le moins privilégié. Cette hiérarchie de privilèges existe dans tous les systèmes d'exploitation actuels. Elle permet plus de stabilité tout en étant multi-utilisateurs et multitâches.

La plupart des systèmes ne disposent que de deux niveaux d'exécution : un pour le mode système et un pour le mode utilisateur. Un processus s'exécutant en mode utilisateur ne dispose pas d'un accès direct au matériel et doit utiliser pour cela les abstractions fournies par le noyau. Il n'a accès qu'à un ensemble restreint de fonctions. Il peut appeler des services du noyau à travers les appels système, fonctions du noyau disponibles pour les applications. Ces fonctions sont regroupées dans l'API et permettent d'effectuer des opérations qui ne sont pas directement disponibles pour les applications (par exemple la gestion des temporisateurs).

Le noyau et les pilotes sont généralement exécutés en mode privilégié, également connu sous le nom de mode superviseur ou mode noyau, et peuvent exécuter ainsi n'importe quelle instruction. Les applications sont plus contraintes dans les instructions qu'elles peuvent exécuter (mode d'utilisateur).

Le mode « utilisateur », destiné aux applications, est un mode non privilégié : certaines instructions sont inaccessibles tout comme une partie de la mémoire. Cela évite qu'une application ne vienne en perturber une autre en modifiant son espace d'adressage. Le mode d'adressage protégé est un des principaux mécanismes de protection de la mémoire. C'est différent quand un procédé d'application exige un certain service du noyau en publiant un appel système (par exemple, l'interruption #0x30 sur *Pentium*, #0x87 sur *PowerPC* ou les *rings* sur *x86*). Le noyau et les pilotes sont exécutés en mode privilégié : toutes les instructions machines sont autorisées et toute la mémoire du système est accessible. N'importe quel comportement anormal dans un pilote est susceptible de modifier l'espace d'adressage du noyau, c'est-à-dire le noyau lui-même.

Ce phénomène est encore aggravé par l'utilisation des langages de programmation de bas niveau, utilisant par exemple lors de la conception des pilotes l'arithmétique des pointeurs et ce, sans gestion intégrée de la mémoire (IMM). La plupart des systèmes d'exploitation courants (par exemple *Linux* et *Windows*XP) disposent d'extensions élaborées avec des langages bas niveau comme C ou C++.

L'espace d'adressage

Un processus peut partager son espace d'adressage avec d'autres processus. Quelques systèmes d'exploitation utilisent un espace d'adressage partagé pour tous les processus, c'est-à-dire sans protection, comme dans le cas du système d'exploitation Dos.

Tous les systèmes d'exploitation ne possèdent pas forcément un noyau protégé des programmes utilisateurs. Les premiers systèmes d'exploitation fournissaient des interfaces avec le matériel directement pour l'exécution des programmes utilisateurs. Cependant, ils ne fournissaient aucun mécanisme de protection contre ces programmes ni même la capacité de protéger certains programmes vis-à-vis d'autres. L'avantage de ces systèmes était leur performance.

L'espace d'adressage correspond à l'isolement virtuel des zones de mémoire consultables par les processus. Une telle protection est habituellement mise en application par une unité de gestion de mémoire (MMU). Chaque processus applicatif est exécuté seulement dans son propre espace d'adressage alors que le noyau et les pilotes partagent un unique et vaste espace d'adressage dans la plupart des noyaux.

Architecture

Les systèmes d'exploitation diffèrent essentiellement par leur architecture interne. En effet, leurs interfaces avec leur environnement, la partie matérielle et la partie logicielle, se présentent souvent de manière similaire. Elles constituent des points de passages intéressants pour étudier leur robustesse par rapport à un environnement défaillant. Nous reviendrons sur ce point au chapitre trois.

Le noyau est le cœur d'un système d'exploitation. Dans le cas d'un noyau monolithique, il fournit la gestion des processus, la gestion de la mémoire, les pilotes de périphériques, les systèmes de fichier, l'accès réseau, etc. Dans le cas contraire, c'est-à-dire lorsque les services d'un système d'exploitation sont mis en œuvre à l'extérieur du cœur du système sous la forme de processus, le cœur ne fournit que les services de base, par exemple la gestion mémoire et le *multithreading*. Le système d'exploitation est alors appelé micronoyau ou même nanonoyau pour les plus petits.

Remarque

Dans les systèmes d'exploitation actuellement disponibles, seuls les termes noyaux monolithiques et micronoyaux sont utilisés, même s'il existe différentes architectures de micronoyaux. Dans la suite, le terme noyau sera utilisé pour désigner la partie du système d'exploitation qui supervise la machine. Le cœur du noyau désignera la partie minimale de ce noyau.

Indépendamment des différentes solutions adoptées pour une famille spécifique de systèmes d'exploitation, deux catégories principales de pilotes peuvent être distinguées :

- Les pilotes d'abstractions logicielles: ils n'ont aucun accès direct à la couche matérielle des dispositifs, mais plutôt à une abstraction (par exemple, pile TCP/IP, système de fichiers).
- Les pilotes de matériel: ils sont concernés par des dispositifs câblés ou non (réseau, disque dur, imprimante, clavier, souris, écran, bus PCI, RAM, etc.).

Ces deux types de pilotes diffèrent sensiblement. Cependant, dans les deux cas, leur rôle consiste à fournir une abstraction d'un dispositif pour le système d'exploitation. Ils peuvent alors interagir.

1.4.2 Exemples de systèmes d'exploitation

Nous allons présenter dans cette partie les architectures les plus répandues dans les systèmes d'exploitation contemporains. Nous verrons des structures compactes comme celle des noyaux monolithiques, favorisant la performance à la modularité du noyau. Nous éluderons les micronoyaux, une architecture très en vogue et très controversée durant les dernières années. L'analyse des exonoyaux, qui constituent une solution alternative aux deux précédentes, clôturera le paragraphe.

N` yau m` n` lithique

Un noyau monolithique de type *Unix* offre tout ce qu'un système d'exploitation nécessite comme services (voir [Bach 1987]): gestion des processus, gestion de la mémoire, communication interprocessus, accès aux périphériques, système de fichiers, protocoles réseau, etc. Les noyaux monolithiques ne mettent pas en œuvre de séparation entre les services qu'ils fournissent, ainsi l'ensemble du noyau s'exécute avec des privilèges élevés. On parle alors, par abus de langage, de processus en mode privilégié. Cependant, par souci de modularité, les noyaux monolithiques actuels permettent, pendant l'exécution, l'insertion et l'élimination de services additionnels. Ces noyaux sont conçus pour les systèmes qui sont redémarrés exceptionnellement et qui n'évolueront pas au niveau des périphériques utilisés.

Certaines améliorations introduites dans *Linux* [Bovet & Cesati 2000], comme l'utilisation de modules insérés dynamiquement, offrent une extensibilité similaire à celle des micronoyaux. Ces extensions s'exécutent en mode noyau et sont donc des sources potentielles d'instabilité. Elles portent les noms de modules sous *Linux* [LinuxHQ 2002]. L'utilisation de modules qui s'exécutent en mode noyau, même pour ceux qui ne nécessitent pas un accès à l'architecture matérielle, pose un problème potentiel de sûreté de fonctionnement. En effet, ils ont accès à toutes les structures de données du noyau et ils s'exécutent avec des privilèges élevés dans l'espace d'adressage du noyau. Les pilotes de *Linux* ont fait l'objet de notre étude expérimentale présentée dans [Albinet *et al.* 2004].

Les micr` n` yaux

Les micronoyaux sont conçus différemment des noyaux monolithiques. Certains services du noyau sont mis en œuvre sous la forme de processus qui peuvent être arrêtés et redémarrés pendant l'exécution. Il en résulte des noyaux plus petits et plus souples. Le système de fichiers, les pilotes de périphériques, la gestion des processus et même une partie de la gestion de la mémoire peuvent être mis en œuvre sous la forme de processus qui s'exécutent « au dessus » du micronoyau. Cette architecture suit un modèle client-serveur. Les processus qui fournissent les services du noyau sont appelés des serveurs. Un processus (client) demande un service au système d'exploitation en envoyant une requête au processus serveur par l'intermédiaire du mécanisme de communication interprocessus (*Inter Process Communication* - IPC). Les micronoyaux peuvent être multitâches, en allouant différents services à différents processus légers[4] d'exécution, offrant ainsi un bon compromis entre fiabilité et performance.

Toutefois le problème majeur des micronoyaux est la mise en œuvre de mécanismes IPC suffisamment rapides. Au début, ces mécanismes ont formé des goulots d'étranglement, mais actuellement, les micronoyaux offrent des mécanismes IPC performants.

Par ailleurs, il faut différencier les micronoyaux de première et de deuxième génération. Les premiers, comme le micronoyau *Mach*, sont assez lourds et fournissent une multitude de services et plusieurs options de réalisation. Les micronoyaux de seconde génération suivent scrupuleusement le modèle de conception d'un micronoyau en offrant un micronoyau avec un

[4] *Threads*

nombre réduit de mécanismes. *L4* [Härtig *et al*. 1997] et *QNX* [Hildebrand 1992] sont des exemples de micronoyaux de seconde génération.

Un des micronoyaux les plus connus est *Windows 2000*. Il met en œuvre une interface matérielle nommée HAL[5] [Solomon & Russinovich 2000]. Les fonctions de cette interface mettent en œuvre la partie dépendante de l'architecture du système d'exploitation, fournissant des fonctions pour accéder au matériel de manière générique. Un micronoyau basé sur *Mach* s'exécute sur la HAL. Sur le micronoyau, et toujours en mode utilisateur, s'exécutent les serveurs offrant les différents services du système : gestion des objets, gestion des entrées/sorties, gestion des processus, système de fichiers, etc. Les pilotes des périphériques sont gérés par le gestionnaire d'entrées/sorties. L'avantage par rapport à *Linux* est que ces serveurs et pilotes de périphériques peuvent s'exécuter dans des espaces d'adressage séparés, offrant ainsi une meilleure protection des données du noyau.

Les ex` n` yaux

L'approche exonoyau a été proposée par le MIT au milieu des années 1990 dans [Hildebrand 1992] puis [Engler *et al*. 1995]. Cette approche consiste en l'utilisation d'un noyau offrant uniquement la protection et le multiplexage des ressources matérielles — la gestion de ces ressources étant du ressort des applications. À l'inverse de *Aegis* et de *Xok* qui proposent des abstractions et des éléments d'ordonnancement processeur qui sont loin d'êtres minimaux, *Think* est un exemple de noyau qui a adhéré strictement à la philosophie exonoyau [Fassino *et al*. 2002]. Il n'impose aucune abstraction du système préalable et réifie de manière très précise les capacités de la machine sous-jacente. Ainsi, les exonoyaux permettent aux programmes utilisateurs de fournir leur propre mise en œuvre des standards exportés habituellement par le noyau. Ceci permet d'améliorer la performance du système global dans la mesure où l'utilisateur a la possibilité de réécrire l'algorithme qui répond le mieux à ses besoins. Cependant, ceci représente une vulnérabilité puisque n'importe quel utilisateur a la possibilité de modifier le code noyau, et ce même si les exonoyaux essayent de forcer les programmeurs à utiliser des langages sécurisés.

Remarques

De toute évidence, de plus en plus d'intérêt est porté à la sûreté de fonctionnement des systèmes d'exploitation dès la phase de conception. La publication assez fréquente sur Internet par les constructeurs de mesures de disponibilité et de fiabilité des systèmes d'exploitation destinés à supporter l'exécution de serveurs de fichiers ou d'applications est un signe de cette évolution. Nous allons voir dans le paragraphe suivant comment l'étalonnage de performance, précurseur de l'étalonnage de sûreté de fonctionnement, de ces systèmes d'exploitation peut s'effectuer.

1.5 Architecture des pil` tes de périphériques

Dans ce paragraphe, nous fixerons un cadre de présentation des interactions entre noyau et pilotes pour les trois systèmes d'exploitation personnels les plus répandus aujourd'hui. Pour

[5] *Hardware Abstraction Level*

commencer, nous verrons le système en vogue de ces dernières années, *GNU/Linux*, système à code source libre. Puis nous nous intéresserons au dernier système-phare de la firme *Microsoft*, *Windows XP*[6]. Enfin, nous aborderons un système d'exploitation fournissant une alternative intéressante à *Windows* et à *Linux*, *MacOS X*, un système avec une personnalité de type *Unix*.

1.5.1 *Linux*

Nous avons présenté dans le paragraphe précédant les systèmes d'exploitation de type *Unix*. Linus Torvalds s'est inspiré d'un de ceux-là, *Minix*, pour créer les bases de *Linux*. Comme dans tous les systèmes de type *Unix*, les périphériques sont représentés sous la forme de fichiers dans le système et sont regroupés dans un répertoire commun. Le principal avantage de cette technique est de permettre un accès « générique » aux pilotes, en réutilisant les techniques d'accès aux fichiers classiques, simplifiant la gestion des composants matériels. Ces fichiers représentant des périphériques sont appelés « fichiers spéciaux ».

Comme le montre la Figure 1-4, les applications peuvent accéder aux périphériques et effectuer

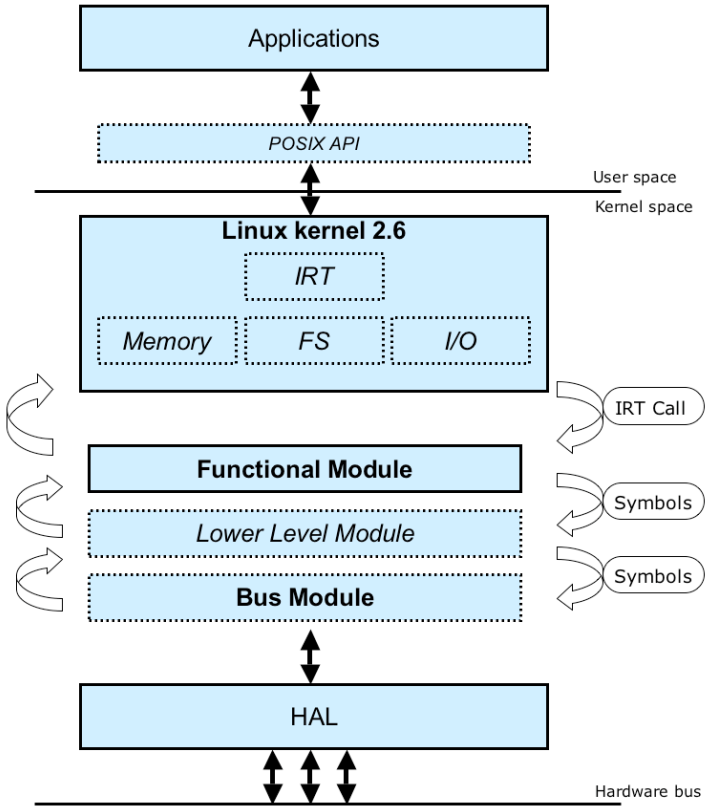


Figure 1-4Architecture des pilotes sous Linux

[6] à l'heure où ce manuscrit est écrit, nous ne disposons pas de suffisamment d'informations sur la dernière version « *Long Horn* » de *Windows*

plusieurs opérations de contrôle Entrées/Sorties (E/S) via l'interface applicative (API).

L'appel au système est alors redirigé grâce à la table des requêtes du noyau (IRT) vers le bon pilote correspondant au matériel demandé et vers la fonctionnalité correspondant à l'appel invoqué (lire ou ouvrir, par exemple). Une routine du pilote correspond alors à cette fonctionnalité et réalise l'opération sur le périphérique via l'HAL (*Hardware Abstraction Layer*). La réalisation de cette routine nécessite souvent l'utilisation de modules inférieurs, de modules liés aux périphériques de transport de données (comme les bus) ou de fonctionnalités du noyau. Les symboles fournissent ces fonctionnalités par le système à travers :

- Son noyau et ses composants : le gestionnaire de fichiers, le gestionnaire d'entrées/sorties, le gestionnaire de mémoire ...
- L'ensemble de ses extensions : pilotes, modules ...

L'insertion et la suppression d'un pilote de gestion de périphériques s'effectuent dynamiquement sous *Linux* grâce à la notion de module. Un module est un composant logiciel qui peut être ajouté en opération au code du noyau. Le redémarrage de la machine n'est pas nécessaire. Aujourd'hui, la plupart des pilotes de gestion de périphériques de *Linux* sont développés sous la forme de module. Ces caractéristiques sont particulièrement intéressantes dans le cas de machines « serveurs », car elles permettent au système de rester disponible lors de modifications matérielles mineures du système.

Chaque pilote contient deux routines pour insérer et supprimer son code du système d'exploitation. Sous *Linux*, elles se nomment *module_init* et *module_exit.* La routine *module_init* est la première routine qu'exécutera le pilote lorsqu'il est chargé. Cette routine comportera un appel à une fonction de chargement dans le noyau, par exemple le symbole *register_chrdev* pour un pilote de type « caractère » *(*voir [Rubini & Corbet 2001]), un nom pour le pilote et l'affectation d'un identifiant. La décharge d'un pilote se conclut par l'exécution d'une routine de type *module_exit*. Elle contient les instructions nécessaires à la libération des ressources et des identificateurs réservés lors du chargement du pilote.

Les autres routines d'un pilote permettant d'agir sur les périphériques à partir d'une application correspondent aux fonctions d'accès aux fichiers :

- *Open* : ouvre le fichier spécial lié au périphérique.
- *Release* : ferme le fichier spécial.
- *Read* : effectue une lecture de données dans le fichier ouvert.
- *Write* : effectue une écriture de données dans le fichier ouvert.
- *Llseek* : déplace le pointeur de lecture dans le fichier spécial.
- *Ioctl* : permet de réaliser une opération de contrôle particulière sur le pilote de périphériques.

Ces opérations sur le fichier spécial du périphérique sont à définir lors du chargement du pilote dans le noyau. Une structure particulière, *file*, est mise à disposition par le noyau pour permettre de référencer tous les points d'accès du pilote.

Pour réaliser ces opérations, les pilotes n'ont pas accès aux bibliothèques C classiques. En effet, par leur position dans le cœur du système d'exploitation, les bibliothèques applicatives ne sont pas disponibles. Néanmoins le module de gestion de périphériques peut-être conduit à utiliser le noyau ou un autre pilote de plus bas niveau pour réaliser, par exemple, des opérations d'écriture sur un bus. Pour faire appel à une autre partie du noyau, les pilotes appellent des fonctions, mises à disposition par le noyau, permettant la réalisation d'opérations complexes.

Ces « symboles » permettent de réaliser diverses opérations comme la gestion de la mémoire, la gestion des interruptions, la gestion du système de fichiers, la gestion du contrôle des transferts de blocs ou l'enregistrement. Par exemple, le transfert de données entre l'espace utilisateur et l'espace noyau peut être réalisé par un des deux couples de symboles suivants : *copy_{from,to}_user* et *{get,put}_user*. Les réservations, vérifications et libérations des ports d'entrée et de sortie s'effectuent respectivement via les fonctions *request_region*, *check_region* et *release_region*. Il existe ainsi une grande variété de fonctions permettant de réaliser l'ensemble des instructions nécessaires à l'accomplissement des opérations du pilote.

Nous décrirons plus précisément ces fonctions dans le chapitre 3.

1.5.2 *Windows XP*

Windows XP est le dernier système d'exploitation « grand public » de la famille *Windows*. Durant de nombreuses années, les systèmes de la famille *Windows* avaient peu évolué. *Windows XP* a toutefois intégré de nouveaux concepts fort intéressants surtout du point de vue de ses pilotes de périphériques. Ces nouveautés sont détaillées dans les travaux présentés dans [Oney 1999] et [Cant 1999].

En effet, bien que basé sur le noyau de *Windows NT*, ce système offre un environnement dédié au développement des pilotes de périphériques sous *Windows* : le *Driver Development Kit* [Microsoft DDK 2002]. Nous aborderons, en fin de chapitre, un des outils de développement du DDK : *Microsoft Driver Verifier*. Déjà présent dans le système d'exploitation *Windows 2000* sous une forme moins complète et moins aboutie, le DDK met à disposition des développeurs un environnement élaboré et accessible de développement des pilotes. Il oriente le cadre de développement des pilotes des périphériques vers une documentation lisible, complète et accessible sur la programmation en mode noyau sous *Windows XP*. Il met également à disposition des outils de développement pratiques permettant la vérification du code lors de la compilation, mais aussi lors de son exécution. Le déboguage et le test du code des pilotes est ainsi plus aisés.

Les pilotes et les périphériques se présentent sous la forme d'object :

- Les périphériques physiques sont des PDO (*Physical Device Object*).
- Les pilotes de périphérique sont des FDO (*Function Device Object*).
- Les filtres des pilotes sont des FiDO (*Filtre Device Object*). Ils permettent le prétraitement d'une requête destinée ou provenant d'un FDO.

Les applications peuvent effectuer plusieurs opérations de contrôle d'entrée/sortie via l'interface applicative de *Windows*, *Win32*. L'appel système est reçu par le gestionnaire I/O du micronoyau. Celui-ci construit une requête d'entrée/sotie appelée IRP (pour *I/O Request Packet*) à destination du pilote associé au matériel demandé. L'IRP indique la fonctionnalité correspondant à l'opération demandée. L'IRP est reçue par le filtre du pilote associé pour un éventuel prétraitement des données. Le filtre « haut » du pilote recompose une IRP à destination du pilote le plus à même de la réaliser (Il peut être différent de la première IRP). Une routine du pilote correspond alors à cette fonctionnalité et réalise l'opération sur le périphérique. Lors de cette exécution, des services sont souvent nécessaires pour réaliser des opérations dans le noyau (écrire sur le bus, gérer les accès concurrents, réserver de la mémoire). Ses services sont réalisés en effectuant des IRP vers d'autres pilotes ou directement vers des fonctionnalités du noyau. Ces IRP peuvent être traitées de nouveau par des filtres avant d'être réalisées par un des services du noyau. La Figure 1-5 illustre ces interactions dans l'architecture des pilotes sous Windows (*Windows Driver Model*-WDM) détaillées dans [Microsoft WDM 2002].

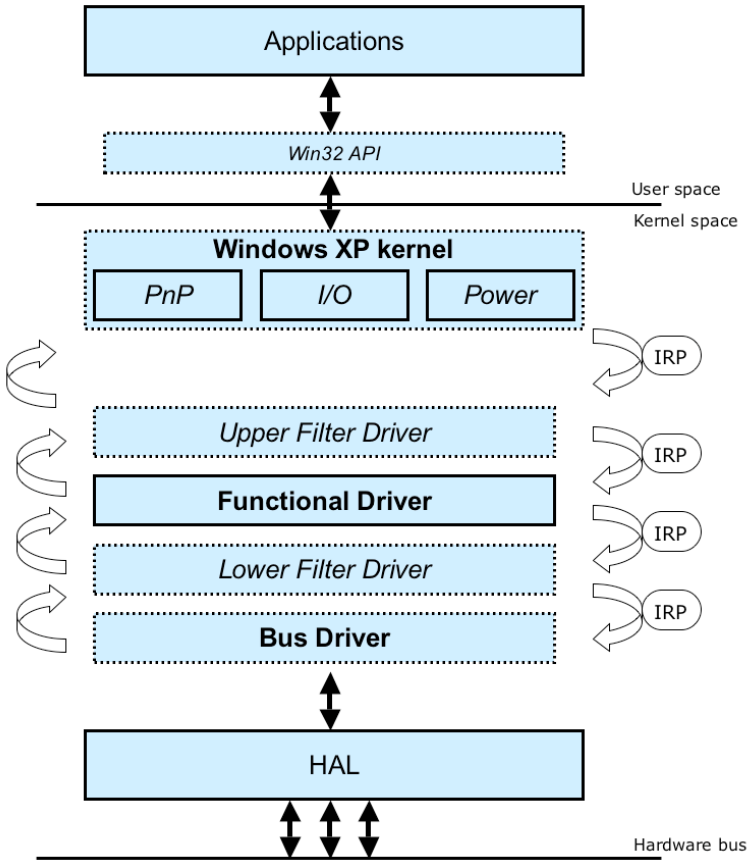


Figure 1-5 Architecture des pilotes sous Windows XP

Une différence avec *Linux* est la représentation des périphériques dans le système. Alors que sous *Linux* les périphériques sont représentés sous la forme de fichiers spéciaux, ils sont représentés sous la forme d'objets sous *Windows*. Tous les pilotes contiennent deux routines pour insérer et retirer le pilote du système d'exploitation. Sous *Windows*, elles se nomment

DriverEntry et *DriverUnload*. *DriverEntry* est la première routine qu'exécute le pilote lorsqu'il est chargé. Cette routine permet d'effectuer les opérations nécessaires au chargement du code du pilote dans le noyau : un nom pour le pilote et la création d'un objet représentant le périphérique. La routine *DriverUnload* est la dernière macro réalisée lors de la décharge du pilote. Elle contient les instructions nécessaires à la libération des ressources et des identificateurs réservés lors du chargement du pilote.

Les pilotes de périphériques *Windows* sont composés également d'un ensemble de routines pour effectuer des opérations d'entrées/sorties et de contrôle du périphérique (écriture, lecture, ouverture, fermeture du PDO). Elles sont pour la plupart similaires à *Linux* et permettent de répondre aux requêtes suivantes :

- *IRP_MJ_CREATE*: ouvre un PDO.
- *IRP_MJ_CLOSE*: ferme un PDO.
- *IRP_MJ_READ*: lit les données d'un PDO ouvert.
- *IRP_MJ_WRITE*: écrit des données dans un PDO ouvert.
- *IRP_MJ_DEVICE_CONTROL*: réalise une opération de contrôle particulière sur le pilote de périphériques (par exemple une *IOCTL)*.
- Diverses IRP : permettent la résolution des fonctionnalités PNP (*Plug And Play*), contrôle de l'alimentation, ...

Pour réaliser ces opérations, les pilotes disposent de routines associées à chacune de ces requêtes. Elles sont développées avec le langage C++, mais ne disposent pas des bibliothèques applicatives communes C, compte tenu de leur emplacement dans le noyau. Un pilote de périphériques nécessite souvent de faire appel à un pilote de plus bas niveau pour réaliser, par exemple, des opérations d'écriture sur un bus. Pour faire appel à une autre partie du noyau, les routines font appel, via une IRP, à des méthodes décrites dans l'environnement DDK du noyau.

Ces méthodes réalisent diverses opérations comme la gestion de la mémoire, la gestion des interruptions, la gestion du système de fichiers, la gestion du contrôle des transferts de blocs ou l'enregistrement. Par exemple, le transfert de données entre l'espace utilisateur et l'espace noyau peut être réalisé par la fonction *RtlMoveMemory*. L'utilisation d'une variable noyau *SystemBuffer* est nécessaire pour obtenir une adresse dans l'espace utilisateur de l'application.

L'insertion et la suppression d'un logiciel de gestion de périphériques s'effectuent dynamiquement sous *Windows XP*. Toutefois, un redémarrage de la machine est souvent nécessaire. Pour un ordinateur particulier, l'influence est très faible, mais il n'en est pas de même pour une machine utilisée comme serveur, par exemple dans le domaine du commerce en ligne.

1.5.3 *MacOS X*

La société *Apple* développe depuis de nombreuses années un système d'exploitation propriétaire : *Mac OS*. La dernière mouture de ce système se base sur un noyau propriétaire de type *Unix* : *XNU*.

Ce noyau se compose de :

- Une partie de *Mach 3.0*, un « micronoyau »[7] issu des premières recherches sur les micronoyaux à l'université de Carnegie Mellon dans les années 80 fournissant les aspects bas niveau du système (gestion des interruptions, protection mémoire, pilotes, une partie de la gestion mémoire, ...).
- Une partie de *FreeBSD* et une autre de *Darwin 7.x* formant un serveur au niveau applicatif fournissant les aspects haut niveau du système (API *POSIX*, pile TCP/IP, VFS, ...).
- Le *I/O Kit* de *Darwin,* qui fera l'objet d'une attention particulière dans ce chapitre compte tenu de son rôle concernant les pilotes de périphériques.

MacOS X se situe entre *Windows* et *Linux* avec un code source partiellement ouvert. De plus, il a bénéficié d'un développement moderne en respect avec certains critères de développement conformes aux technologies recommandées dans [Bershad *et al.* 1995].

En effet, le développement des pilotes de périphériques sous *MacOS X* est réalisé grâce à son *Kernel Development Kit* comprenant l'*I/O Kit* de *Darwin*. Il a permis une avancée conséquente dans la programmation noyau de *MacOS X*. Il propose un environnement de développement moins évolué que le DDK de *Microsoft*, dû à l'absence de certains outils permettant le test et le déboguage du pilote de périphérique. Toutefois, il dispose d'une riche bibliothèque de classes de pilotes avec un ensemble de fonctions et classes en C++ embarqué (une version allégée du langage), y compris des abstractions orientées objet communes à diverses familles de pilotes. En outre, le *I/O kit* fournit une interface pour les périphériques permettant à une application de communiquer et de commander un dispositif directement depuis l'espace utilisateur.

L'insertion et la suppression d'un logiciel de gestion de périphériques s'effectuent dynamiquement sous *MacOS X*. Un redémarrage de la machine n'est pas nécessaire pour les pilotes qui ne sont pas liés au matériel essentiel au fonctionnement du système.

Les périphériques sont représentés sous la forme d'objets appelés *Nub*. L'*I/O Kit* fournit une abstraction haut niveau de chaque périphérique et des canevas pour les pilotes associés. Les méthodes disponibles dans un pilote sont sensiblement les mêmes que pour les deux systèmes précédents.

[7] Les micronoyaux modernes sont de taille très inférieure à *Mach.*

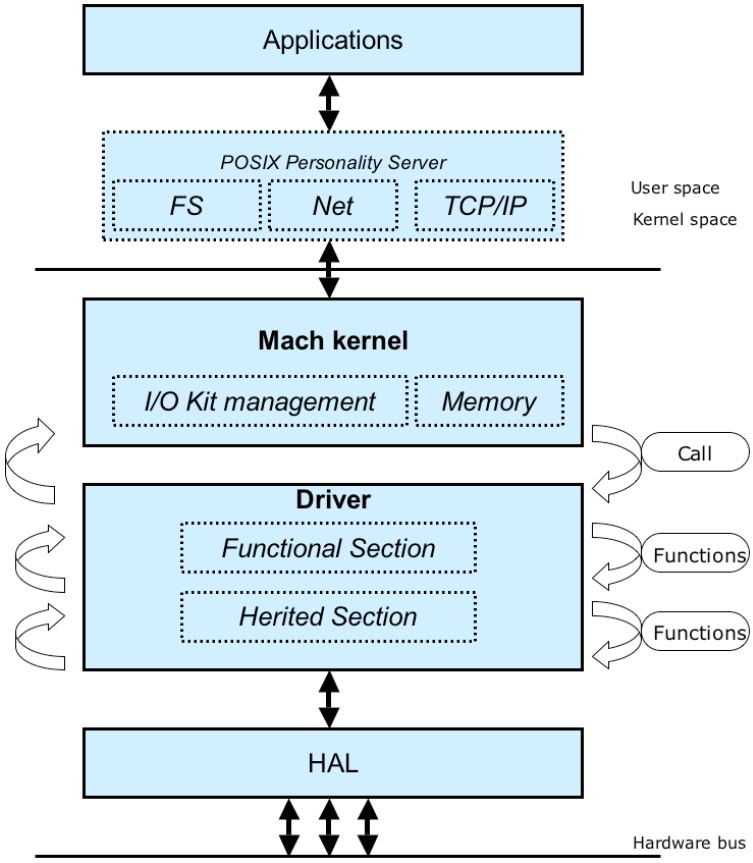


Figure 1-6 Architecture des pilotes sous MacOS X

Comme le montre la Figure 1-6, les applications peuvent accéder aux périphériques et effectuer plusieurs opérations de contrôle Entrées/Sorties (E/S) via l'interface applicative (API) de *MacOS X*. L'appel système est alors redirigé via le module de gestion *I/O Kit* vers le pilote correspondant au matériel demandé. La routine du pilote correspondant à l'appel invoqué est alors exécutée et réalise l'opération demandée sur le périphérique. Lors de cette exécution, des services noyau sont souvent nécessaires (écrire sur le bus, gérer les accès concurrents, réserver de la mémoire, gérer les accès concurrents). Ils sont réalisés en invoquant des méthodes du *I/O Kit*. Ces méthodes peuvent être fournies, à l'origine, par un pilote différent ou par le noyau.

Une des particularités de *MacOS X* se situe au niveau de l'emplacement des couches d'abstraction des ressources matérielles (HAL). En effet, sous *MacOS X*, de nombreuses ressources matérielles sont partagées au niveau applicatif comme le montre la Figure 1-7. Les applications peuvent interagir avec le matériel grâce à l'utilisation de l'*I/O Kit*. Celui-ci fournit à la fois l'abstraction logicielle du matériel aux pilotes (par exemple *Audio HAL* sur la figure) et aux applications (*Audio family*). Une application peut donc agir spécifiquement sur un périphérique. Cependant, cette architecture ne modifie pas la vision globale de l'architecture des pilotes dans les systèmes modernes. En effet, sous *MacOS X*, l'*I/O Kit* reste l'interface entre les pilotes et le reste du système.

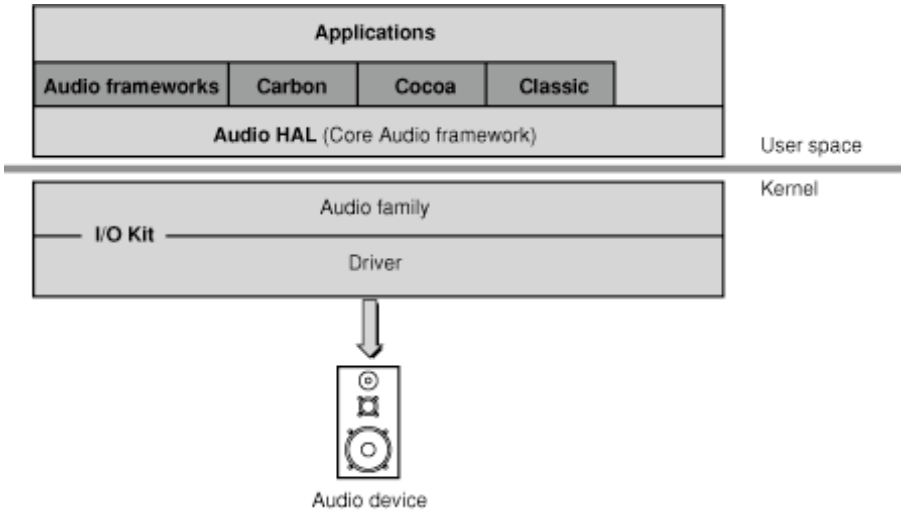


Figure 1-7 Architecture des ressources audio sous MacOS X

1.5.4 Remarques

Certains systèmes d'exploitation modernes ont sensiblement amélioré l'architecture logicielle et l'environnement de développement concernant les pilotes de périphériques afin de permettre une intégration plus rapide et plus efficace des nouveaux dispositifs. Pour chacun des trois systèmes d'exploitation considérés, l'architecture présente de fortes similarités. Les applications effectuent des requêtes sur un matériel au niveau de l'API vers le noyau. Ce dernier redirige l'appel vers le logiciel pilotant ce matériel en spécifiant la fonctionnalité demandée par l'application. La routine correspondante du pilote est alors exécutée.

Les pilotes de périphériques, comme toutes les extensions, se situent dans l'espace d'adressage du noyau du système. Ils ne peuvent donc pas utiliser les bibliothèques classiques disponibles pour la programmation d'applications classiques. Pour permettre la réalisation de leurs fonctionnalités, ils ont besoin d'accéder à des fonctionnalités du noyau ou celles d'autres pilotes. Par exemple pour l'enregistrement du pilote, celui-ci doit faire appel au service du noyau correspondant à travers des fonctions réservées (comme *reg_X* sous *Linux* et *register_X* sous *Windows)*. Pour une réservation d'espace mémoire, le pilote doit faire appel à la gestion de mémoire du système d'exploitation. Lors d'une écriture dans un composant matériel, le pilote de périphériques peut aussi avoir besoin d'un pilote de plus bas niveau, comme celui du périphérique bus. L'ensemble de ces appels est réalisé via des fonctions du noyau. Ce sont ces fonctions qui constituent la partie visible (dans le concept de « boîte noire ») du noyau du point de vue des extensions.

1.6 Amélioration de la sûreté de fonctionnement

Durant les dernières années, de nombreux travaux ont eu lieu afin de prendre en compte les problèmes de sûreté de fonctionnement liés aux pilotes de périphériques dans les systèmes d'exploitation commerciaux. Les universitaires et les industriels ont développé des techniques visant à améliorer les problèmes liés aux pilotes de périphériques dans les systèmes. Les

principaux travaux ont porté sur la compréhension et la résolution des problèmes de sûreté de fonctionnement pouvant conduire à des terminaisons brutales du noyau ou de ses applications.

Dans [Hartig *et al*. 1997] et [Engler *et al*. 1995], les recherches se sont intéressées, entre autres, à la minimisation des effets des fautes présentes dans les pilotes de périphériques en proposant une architecture du système d'exploitation différente des systèmes traditionnels monolithiques. Les auteurs proposent dans [Bershad *et al* 1995] et [Réveillère & Muller 2001] d'utiliser des langages sûrs lors de la conception du noyau ou d'un pilote pour prévenir les fautes. D'autres études se sont portées sur l'amélioration de la sûreté de fonctionnement des logiciels [Seltzer *et al* 1996] ou du matériel [Chiueh *et al* 1999] pour permettre la détection et la tolérance de fautes provenant des pilotes. La disponibilité d'un système ne dépend pas seulement de l'analyse de la défaillance, mais aussi du rétablissement rapide du service après une interruption. Par conséquent, le système d'exploitation ne doit pas simplement isoler les modules de gestion de périphérique défectueux, mais il doit aussi leur permettre de reprendre rapidement leur service, après une remise en marche ou une procédure de récupération. Les techniques de recouvrement des problèmes des dysfonctionnements du matériel sont abordées dans [Edwards & Matassa 2002]. Il faut alors détecter et isoler rapidement les effets d'une défaillance afin de procéder à la récupération d'un état antérieur « stable ».

Nous allons présenter ces différentes techniques d'amélioration de la sûreté de fonctionnement.

1.6.1 Amélioration de l'architecture du système

Afin de réduire l'impact de la défaillance d'un pilote de périphérique, une des premières solutions envisageables serait de les empêcher d'exécuter des instructions privilégiées. En effet, contrairement aux instructions classiques, les instructions privilégiées rendent le noyau directement vulnérable aux erreurs. La possibilité de limiter les instructions possibles pour les extensions permettrait de limiter la propagation des fautes dans le reste du système et diminuerait l'impact de celle-ci, comme c'est déjà le cas pour les logiciels applicatifs.

Le principal problème provient de la perte de performance liée aux changements de contexte supplémentaires générés par une telle situation. En effet, dans la plupart des systèmes modernes, lorsque qu'un pilote souhaite accéder à une ressource matérielle, il peut le faire directement, sans faire un changement de contexte pour mettre le processeur en mode superviseur. Or, si on limite les privilèges d'exécution d'un pilote, il doit alors réaliser un changement de contexte avant d'exécuter des requêtes sur le matériel puis revenir à son mode initial, un mode non privilégié. Cette activité supplémentaire semble négligeable dans le cas d'une application classique, mais pour un pilote, qui effectue constamment des opérations sur le matériel, le nombre d'accès aux ressources matérielles est considérable et présente une lourde part de la charge de travail. L'ajout de ces deux changements de contexte (aller et retour) alourdit sensiblement les performances du système.

Un autre inconvénient de cette technique vient de l'enrichissement nécessaire de l'interface applicative pour permettre la réalisation des opérations « en rapport avec le matériel ». En effet, l'abstraction des ressources matérielles est alors nécessaire au niveau de l'API, alourdissant la portabilité du système. Pour que l'extension en mode utilisateur puisse accéder aux ressources, il lui faut la participation du noyau. Cela n'implique pas seulement des

changements de contexte entre le mode noyau et le mode utilisateur, mais également la nécessité de réaliser des abstractions physiques pour le noyau.

Architecture micr` n` yau

Comme nous l'avons présenté au début de ce chapitre, les premières générations de micronoyaux (voir Figure 1-8) souffraient d'être lentes et peu flexibles.

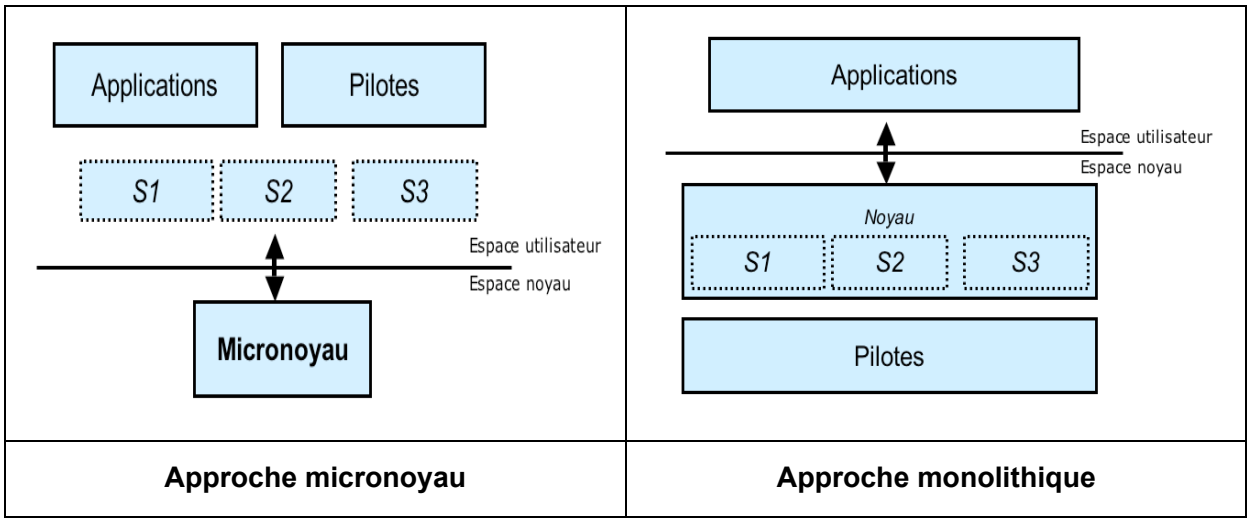


Figure 1-8Différentes approches d'architecture système

Les services (*S1*, *S2* et *S3*) présent dans le noyau dans l'approche monolithique, sont mis en œuvre sous la forme de processus applicatif dans l'approche micronoyau. De fait, rien ne justifie la présence des pilotes dans l'espace du noyau. Ils peuvent en effet bénéficier des services du système sans changement de contexte comme une application classique. Cependant, la performance globale du système dépend alors en grande partie des mécanismes IPC, souvent débordés par la quantité d'échanges à effectuer.

Le micronoyau *L4* semble prouvé que ce n'est pas inéluctable. *L⁴Linux*, un micronoyau *L4* sur lequel le système *Linux* a été porté, semble fournir des performances « proches » d'un *Linux* natif et meilleures que la version micronoyau *MkLinux*. En effet, les expériences présentées dans [Hartig *et al*. 1997] montrent que ce système garde de fortes propriétés d'extensibilité tout en gardant des performances acceptables. Le résultat le plus intéressant, dans le contexte de nos travaux, repose sur l'évaluation de la perte de performance liée à l'insertion des pilotes :

- Soit directement dans le noyau,
- Soit comme une application, en mode utilisateur.

Dans *L⁴Linux*, les pilotes disposent de privilèges d'exécutions limités, toutefois, ses performances sont meilleures que certains systèmes sans limitation de privilèges (*MkLinux*). L'inconvénient de *L⁴Linux* se situe au niveau du bloc que compose le serveur *Linux* au niveau applicatif. En effet, dans un micronoyau classique comme *Hurd,* les services fournis par le système au niveau applicatif sont des serveurs « indépendants ». Cette solution est considérée comme une piste prometteuse pour remplacer *Mach* et *Linux* dans GNU.

Architecture ex` n` yau

Diverses structures de logiciels d'exploitation ont été proposées pour répondre aux besoins actuels : les systèmes avec un noyau monolithique, ceux basés sur un micronoyau, les systèmes d'exploitation virtuels ainsi que les systèmes d'exploitation dédiés.

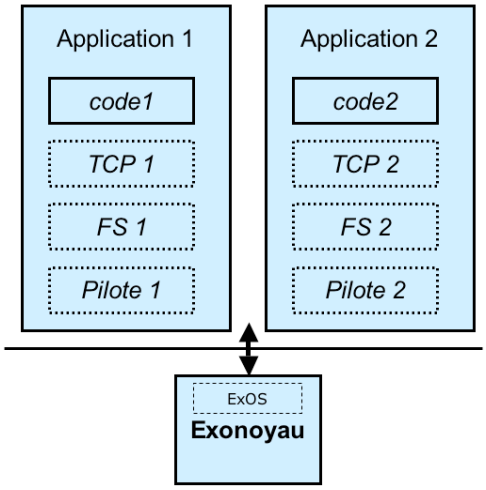


Figure 1-9 Approche Exonoyau selon XOK

Un exonoyau s'écarte nettement des travaux conventionnels des systèmes d'exploitation car il ne fournit pas les abstractions matérielles sur lesquelles les applications sont traditionnellement établies. Au lieu de cela, il se limite à transmettre les mécanismes « haut-niveau » du matériel. Des primitives basiques, des bibliothèques et des serveurs d'application peuvent directement utiliser les abstractions traditionnelles du système d'exploitation, de manière à les adapter à leurs besoins. L'architecture du système basé sur l'exonoyau XOK, présenté dans [Engler *et al.* 1995], est illustré dans la Figure 1-9. Actuellement, il ne fonctionne que sur une architecture *x86*. ExOS est la bibliothèque utilisateur système fournissant une implémentation extensible d'un système d'exploitation de type *Unix*. Les applications emploient cette bibliothèque pour interagir avec les ressources matérielles.

Le principal inconvénient de cette approche réside dans le fait que le concepteur de l'application doit implémenter dans son application les services traditionnellement présents dans le noyau ainsi que les pilotes qui interagiront avec le matériel. Le développement d'une application devient alors un procédé complexe et généralement les exonoyaux ne supportent qu'une application à la fois (d'où la formule « systèmes dédiés »).

1.6.2 Amélioration de la phase de conception

Les langages spécifiques aux domaines (DSL pour *Domain Specific Language*) sont basés sur la confiance dans la machine virtuelle et le compilateur pour détecter et corriger les erreurs dans les pilotes lors de leur conception. L'objectif est de programmer uniquement des pilotes respectant des critères stricts de sûreté de fonctionnement. Des langages « sûrs » comme Java peuvent prévenir les pilotes d'un accès intempestif à la mémoire du noyau, grâce à leur mécanisme intégré de protection de la mémoire. Réaliser les spécifications d'un logiciel avec un langage de type DSL permet d'améliorer la réutilisation des expertises, grâce, entre autres,

aux exigences de qualité d'un cahier des charges clair et précis. L'utilisation d'une bibliothèque prédéfinie permet, de plus, une meilleure réutilisation du code. Enfin, l'amélioration de la fiabilité du code est accrue par des propriétés de vérification et les mécanismes de prévention d'une utilisation incorrecte de la bibliothèque. La productivité lors de la phase de développement est ainsi améliorée.

Le principal inconvénient d'utiliser un langage spécifique dans un COTS est la nécessité de réécrire les pilotes existants, ce qui représente une lourde charge de travail et de temps. L'utilisation d'un DSL lors de la réalisation d'un logiciel est généralement associée à un développement avec un langage sûr (de type Java ou Modula-3). Ces langages introduisant généralement une machine virtuelle comme support d'exécution, ses mécanismes de vérification créent une charge supplémentaire lorsque le pilote effectue des opérations (par exemple, un transfert de données). Il n'y a pas actuellement de mécanisme simple et efficace pour accéder à un pilote avec des modes d'accès à typage « sûr ».

DEVIL

DEVIL (*DEVice Interface Langage*) est un IDL[8] destiné à produire la couche inférieure d'un module de gestion de périphérique, c'est-à-dire l'interaction basique avec le périphérique dont les principes sont présentés dans [Réveillère & Muller 2001]. Des spécifications estampillées DEVIL décrivent rigoureusement les mécanismes d'accès, le type et la disposition de données échangées lors de la manipulation du périphérique, ainsi que ses propriétés comportementales (voir Figure 1-10). La partie basse du pilote dépend uniquement de l'architecture matérielle et peut être utilisée sur différents systèmes d'exploitation.

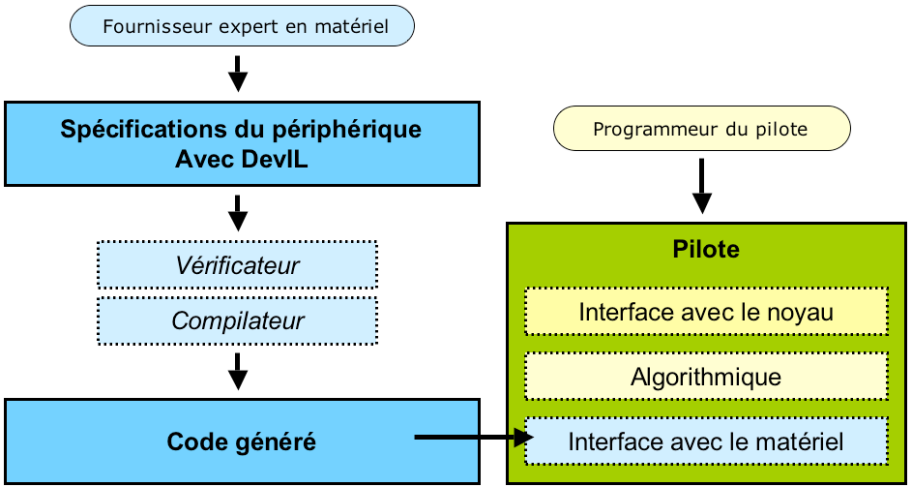


Figure 1-10 Conception d'un pilote avec la méthode DEVIL

Les spécifications fournies par un expert en matériel permettent, grâce au langage DEVIL, de générer le code réalisant l'interface avec le dispositif. La programmation du pilote est alors allégée au niveau de développement de la couche basse liée au matériel. Les pilotes obtenus

[8] IDL : "*Interface Definition Language*", langage permettant de décrire une interface

grâce à DEVIL ont des performances proches des pilotes d'origine tout en limitant fortement les erreurs résiduelles dans le code du pilote (environ 70%).

1.6.3 Amélioration par recouvrement du noyau

Concernant les pilotes, les outils d'empaquetage des éléments du noyau (*kernel wrapping*) ont pour objectif de tester les modules de gestion de périphériques lors de leur intégration au système. Pour cela, ils soumettent les pilotes à une charge de travail nécessaire à l'activation du maximum de leurs fonctionnalités. Lors de la réalisation de ces tâches par le pilote, celui-ci invoque des fonctions du noyau. Ce sont ces appels que l'outil vérifie afin de contrôler l'intégrité de leurs paramètres.

Le recouvrement du noyau est permis si tous les appels entrant et sortant des modules de gestion de périphérique sont effectués avec un code spécifique (HAL ou DDI pour *Windows*), permettant de suivre l'utilisation des ressources et de vérifier certaines conditions d'exécution. Il peut s'assurer de l'allocation de la mémoire, ainsi que de la gestion des interruptions.

Microsoft Driver Verifier

Microsoft Driver Verifier est un outil en mode noyau permettant de surveiller les modules de gestion de périphérique de *Windows*. *Microsoft* encourage fortement les fabricants de matériel à examiner leurs pilotes avec ce vérificateur pour s'assurer qu'ils effectuent des appels légaux de fonctions sans danger pour le système. Le vérificateur de pilote a été enrichi avec de nouveaux essais et dispositifs pour *Microsoft Windows XP*. Cependant, le recouvrement de noyau ne peut pas empêcher un pilote de corrompre accidentellement le système d'exploitation par certaines opérations, comme lors d'une écriture via un pointeur.

1.6.4 Amélioration matérielle

Certaines recherches ont pour objectif la définition d'une architecture d'amorçage fiable et sécurisée présentant une résolution des problèmes de sûreté de fonctionnement via le matériel.

Palladium

Dans [Chiueh *et al.* 1999], le mécanisme *Palladium* cherche à prévenir, à travers une combinaison d'outils matériels et logiciels propre à l'architecture *X86*, une mauvaise utilisation de la mémoire. La protection de la mémoire virtuelle est utilisée pour isoler des problèmes de corruption de données, un des défauts fréquents relevé dans les pilotes, et prévenir l'écriture dans un mauvais segment de données.

Les auteurs montrent que les mécanismes de protection proposés conduisent à une faible perte de performance pour les transferts d'informations importants. Malheureusement, ils ne peuvent pas déceler certaines erreurs, comme celles qui sont provoquées par le masquage inexact d'interruptions. De plus, la perte de performance lors de l'écriture d'un ensemble restreint de données peut s'avérer importante (relativement à la taille des données).

Il est important de noter que ce mécanisme n'a rien à voir avec la proposition homonyme controversée de Microsoft pour, selon ses dirigeants, « améliorer la sécurité des systèmes informatiques ».

1.6.5 Amélioration du logiciel

La technique d'Isolation des Fautes Logicielles (SFI[9]) enrichit le code à traiter, insérant de multiples points de vérification lors de l'exécution du code visant à limiter les effets des fautes. La cible est le code binaire, la technique est donc indépendante du langage source et du compilateur. Elle a pour objectif de vérifier les accès au code et aux données, ainsi que la validité des adresses mémoires en se fiant à l'architecture sous-jacente.

La technique SFI fournit la plupart des avantages procurés par un changement de privilège concernant la limitation des effets des fautes. Elle est difficile à implémenter quand les plages d'adresses accessibles sont non contiguës, ce qui est généralement le cas dans les systèmes d'exploitation modernes. Le code SFI étant indépendant de l'architecture sous-jacente, les transitions avec une partie non SFI sont économiques. Toutefois, la présence de mécanismes supplémentaires dans le code exécutable entraîne certaines lenteurs.

Vino

Un des préceptes du système d'exploitation *Vino* [Seltzer *et al*. 1996] est que le code installé par un pilote du noyau ne doit pas compromettre l'intégrité du système. En conséquence, l'installation de nouvelles pilotes (pilotes de périphériques et modules inclus) se réalise avec une attention particulière.

Vino utilise du code SFI comme mécanisme de sûreté de fonctionnement couplé avec un système de transactions pour gérer les ressources. La combinaison de ses deux mécanismes permet d'éviter une grande partie des défaillances des pilotes. Les mécanismes de SFI (ainsi que des techniques cryptographiques pour protéger cette étape face aux malveillances) s'assurent que le code ajouté ne pourra pas faire :

- Référence à des données protégées ou inexistantes,
- Des appels invalides,
- Un masquage des compteurs d'instruction du processeur

Il est également nécessaire de valider n'importe quelles données entrantes et sortantes du code du pilote :

- Les appels réalisés par le pilote vers le noyau,
- Le retour renvoyé par le pilote à la fonction qu'il a appelé.

La valeur de retour est vérifiée. En effet, cela évite que le pilote bloque le système en traitant une page inexistante. Afin de garantir que toutes les données produites par le pilote puissent être contrôlées, toute fonction externe du pilote doit être déclarée.

Le système de transaction permet d'effectuer un traitement de nettoyage efficace lorsqu'un pilote effectue des opérations inacceptables. Le pilote et ses opérations (réservation mémoire)

---

[9] Software Fault Isolation

peuvent alors être totalement éliminés du système en profitant d'un retour arrière des transactions effectuées avec le pilote.

Nooks

Un « *nook* » est un environnement protégé pour l'exécution de pilotes. En 2003, la technique *Nooks* propose une solution pour que les systèmes d'exploitation puissent supporter l'exécution de pilotes dans un environnement SFI recouvrable. L'objectif est d'éviter qu'un pilote défaillant empêche le reste du système de fonctionner. De plus, le système d'exploitation résultant offre différents niveaux d'isolation et d'exécution adaptables aux différents pilotes. L'objectif de *Nooks* est de fournir ces dispositifs sans changer l'architecture du système d'exploitation et la gestion des modules de gestion de périphérique, pour maximiser la compatibilité avec l'existant. Les pilotes peuvent partager un environnement pour des raisons d'optimisation. *Nooks* s'interpose entre les périphériques et les pilotes via des interruptions et, selon le niveau de sûreté requis, en émulant l'accès à la mémoire mappée des registres du périphérique. De plus, *Nooks* intercepte les appels entre le noyau et le pilote, ce qui permet de vérifier l'utilisation des ressources du système et les échanges de données. Les pilotes s'exécutent toujours dans l'espace d'adressage du noyau mais dans différents domaines de protection. *Nooks* interdit à un pilote d'accéder directement à une adresse n'étant pas associée à son domaine. Le système *Nooks* a été mis en œuvre sur *Linux* très récemment dans [Swift *et al.* 2003] et [Swift *et al.* 2004]. Les résultats montrent une amélioration importante de la fiabilité du système d'exploitation évalué, en détectant et en recouvrant la plupart des fautes pouvant conduire au blocage du système. L'insertion de *Nooks* dans un système entraîne peu de modifications sur le code des pilotes et du système. La surcharge (*overhead*) lors de l'exécution varie en fonction des pilotes considérés (de 1% à 50% d'utilisation processeur en plus).

1.6.6 Remarques

Le Tableau 1-1 compare les différentes techniques mises en œuvre durant les dernières années dans l'objectif d'améliorer la sûreté de fonctionnement dans les systèmes d'exploitation liée aux pilotes de périphériques.

Elles sont classées selon différents critères respectés ou non (respectivement « ✔ » ou « ✗ »):

- Le respect de la performance du système,
- La disponibilité actuelle de la méthode,
- La facilité de la mise en œuvre
- L'efficacité en termes de prévention, détection et tolérance aux fautes affectant les pilotes.

Par respect de la performance, nous entendons les capacités du système sans les mécanismes de prévention et tolérances aux fautes liés aux pilotes. Une faible perte de performance sera considérée comme un facteur positif de la méthode. Les systèmes originaux comme *L*^{4}*Linux* et *Vino* sont considérés comme conservant une performance acceptable malgré les mécanismes supplémentaires mis en œuvre.

Dénomination	Technique	Respect de la performance	Disponibilité	Facilité à implémenter	Efficacité
L4Linux	Modification des privilèges	✔	✗	✗	✔
XOK	Modification des privilèges	✗	✗	✗	✔
Devil	Langages spécifiques	✔	✔	✗	✔
Driver Verifier	Recouvrement du noyau	✔	✔	✗	✗
Palladium	Protection matérielle	✔	✔	✔	✗
Vino	Isolation des fautes par logiciel	✔	✔	✗	✔
Nooks	Isolation des fautes par logiciel	✔	✔	✔	✔

Tableau 1-1 Approches liées à la prévention et la tolérance aux fautes liées aux extensions

La disponibilité d'un système évalue sa mise en opération dans les domaines modernes à forts besoins en sûreté de fonctionnement. La disponibilité d'une technique dépend de sa portabilité aux systèmes les plus répandus actuellement. Nous évaluons également les possibilités que la méthode soit reprise par ces systèmes. Le projet *Palladium* est applicable sur toutes les plates-formes matérielles et la technique présentée dans *Nooks* peut être adaptée à de nombreux systèmes d'exploitation du marché.

La facilité de mise en œuvre évalue l'investissement nécessaire pour l'application de ces techniques à un système. Elle dépend entre autres de la disponibilité.

Enfin, l'efficacité représente le gain par rapport aux besoins actuels en sûreté de fonctionnement des systèmes d'exploitation. En effet, les implémentations ayant pour objectifs la tolérance et la prévention des fautes matérielles sont, pour le moment, jugées moins en adéquation avec les fautes réellement observées en opération. Les techniques portant sur la prévention de fautes logicielles seront donc considérées comme plus efficace par rapport aux besoins actuels. C'est le cas du produit de la firme *Microsoft*, *Driver verifier*.

1.7 C` nclusi` n

Les études récentes sur les systèmes d'exploitation, notamment ceux de la famille *Windows* et *Linux,* montrent que les modules de gestion de périphérique sont la principale cause des défaillances des systèmes d'exploitation. Nous avons vu l'importance du code des pilotes dans le système ainsi que la difficulté inhérente à un développement dans le noyau. Les connaissances nécessaires à une bonne conception et les contraintes de développement d'une extension handicapent la qualité en termes de sûreté de fonctionnement des programmes

pilotant des périphériques. Cela fait des pilotes l'enjeu actuel le plus important de la sûreté de fonctionnement des systèmes d'exploitation.

En effet, l'amélioration des systèmes d'exploitation dépend aujourd'hui des logiciels de gestion des périphériques du système. Le noyau n'est plus la source principale des bogues. Les différentes techniques qui ont été mises en œuvre durant les dernières années ont permis de nets progrès en termes de compréhension et lutte contre les problèmes de sûreté de fonctionnement liés aux pilotes. Elles doivent permettre de tolérer et éviter certaines fautes provenant des extensions.

Les travaux pour résoudre les problèmes de sûreté de fonctionnement liés aux pilotes de périphériques dans les systèmes d'exploitation commerciaux exigent souvent de profondes modifications et un fort investissement (humain et financier). De telles approches sont aujourd'hui économiquement impossibles sur les pilotes de périphérique, qui sont les extensions les plus communes de logiciels d'exploitation et représentent un investissement énorme en temps d'élaboration. Refaire le développement de l'ensemble des pilotes de périphériques d'un système représenterait un travail et des coûts colossaux. La modification structurelle d'un système entraînerait aussi des effets de bords inacceptables (perte de compatibilité ...) ; par conséquent aucune de ces approches n'a été réussie à ce jour dans un système commercial. Cependant, les prochains systèmes pourraient intégrer de telles technologies. Cela supposerait des contraintes fortes en terme de développement de pilotes de périphériques et freinerait certainement l'installation du nouveau système dans le marché. Un système d'exploitation est en effet plus attractif lorsque la plupart des pilotes « du marché » sont disponibles. À l'avenir, il est clair que l'amélioration de la sûreté de fonctionnement des logiciels d'exploitation dépendra des modules de gestion de périphérique.

Dans ce contexte, la nécessité d'évaluer un système, en prenant en compte ses efforts vis-à-vis de la sûreté de fonctionnement du point de vue de ses extensions, est une des principales motivations de nos travaux car les concepteurs de systèmes cherchent actuellement des solutions pour :

- L'évaluation des solutions possibles pour résoudre les problèmes de sûreté de fonctionnement du point de vue des pilotes.
- L'évaluation globale des systèmes d'exploitation pour, par exemple, valider l'intégration d'un COTS dans un système informatique plus important.
- L'évaluation de COTS candidats à une intégration en vue d'une comparaison des alternatives possibles.

Des travaux ont été réalisés dans le cadre de la sûreté de fonctionnement pour définir un cadre pour l'évaluation de la sûreté de fonctionnement des systèmes d'exploitation. Nous les présenterons dans le prochain chapitre.

Chapitre 2 Étal` nnage de sûreté de f` ncti` nnement

Pour des raisons essentiellement économiques, la tendance actuelle est à la réutilisation des composants logiciels (commerciaux ou non) au sein des systèmes d'exploitation. En effet, lors du développement d'un système, la possibilité d'intégrer des fonctionnalités « clefs en main » (COTS) permet de mieux maîtriser les coûts de développement. De plus, cela permet de choisir le composant le plus adapté afin de répondre au mieux aux besoins du système. Pour choisir et comparer des solutions, le concepteur de systèmes doit disposer de données évaluant chacune d'elles, faisant de l'évaluation une étape cruciale dans le processus d'intégration des composants. Pour cela, chaque composant doit être « étalonné » grâce à des mesures correspondant aux critères recherchés.

Aujourd'hui, la performance n'est plus le seul critère de sélection, la sûreté de fonctionnement l'est tout autant. En nous basant sur les étalons de performance existants, nous essayons d'apporter une solution pour évaluer la sûreté de fonctionnement des systèmes d'exploitation. Dans ce contexte, nos travaux ciblent plus particulièrement le point de vue des extensions et des pilotes de périphériques. Les étalons actuels permettent de comparer les performances des différents systèmes grâce à des mesures comme la vitesse d'exécution ou les délais de réponse. D'une manière similaire, les étalons de sûreté de fonctionnement fournissent des mesures spécifiques afin de pouvoir choisir des systèmes ou composants candidats en fonction des critères de sûreté de fonctionnement. Cette thèse s'intègre dans le projet DBench présenté dans [Kanoun *et al*. 2002] et [Madeira *et al*. 2002a], dont l'objectif est de fournir un cadre conceptuel et un environnement expérimental pour l'étalonnage de sûreté de fonctionnement de COTS ou de systèmes à base de COTS.

Le but de ce chapitre est d'établir le cadre d'évaluation de la sûreté de fonctionnement de COTS et de systèmes informatiques à base de ces composants. Dans un premier temps, nous établissons les concepts généraux concernant les systèmes d'exploitation pour l'élaboration d'un cadre pour leur étalonnage. En effet, un étalon est avant tout un standard portable; Son objectif est de permettre de comparer les résultats obtenus avec différentes solutions (différents systèmes d'exploitation ou différents composants). Nous présentons les similarités et les différences des architectures les plus courantes puis nous présentons les étalons de performance, qui ont servi de base au cadre conceptuel des étalons de sûreté de fonctionnement. Nous abordons ensuite la définition d'un étalon de sûreté de fonctionnement par ses dimensions de catégorisation et ses mesures, sa mise en œuvre est ensuite discutée une fois ces caractéristiques définies. Nous poursuivons par une présentation des différents travaux sur la caractérisation de systèmes d'exploitation par rapport à un critère d'importance : l'origine, externe ou interne au système, des fautes que l'on désire étudier. Nous présentons les travaux déjà réalisés touchant à la caractérisation des systèmes d'exploitation vis-à-vis du

noyau ou de ses extensions. Nous aborderons plusieurs approches parmi lesquelles l'injection de fautes, la méthode analytique et une technique liée au contrôle de flux.

2.1 Étal` n de perf` rmances

L'un des premiers critères de sélection d'un système d'exploitation est sa performance. Les étalons de performance constituent un bon moyen pour l'évaluer. Nous avons identifié dans la littérature trois classes d'étalons pour déterminer la performance d'un système d'exploitation : le *profiling*, les macro-étalons et les micro-étalons.

Nous détaillons des étalons de performance dédiés aux systèmes d'exploitation que nous avons jugés pertinents pour les étalons de sûreté de fonctionnement. L'objectif des macro-étalons est de fournir une mesure de performance globale du système. Les micro-étalons permettent de caractériser la performance des mécanismes internes du système. Certains de ces étalons sont aujourd'hui des standards. Inspiré par cette expérience, nous avons défini un cadre pour les étalons de sûreté de fonctionnement.

Profiling

Le *profiling* est une technique qui nécessite généralement l'instrumentation du code source du système cible pour insérer des balises. Pendant l'exécution et à chaque balise, un certain nombre d'informations pertinentes est collecté. Un exemple d'outil de *profiling* est détaillé dans [Yaghmour & Dagenais 2000]. Il est à noter que certains noyaux mettent en œuvre un outil de *profiling* interne. Cette technique s'avère souvent très intrusive et nécessite une fine connaissance du noyau. La modification du système évalué n'est pas compatible avec les critères d'étalonnage. En effet, l'étalonnage a pour objectif l'obtention d'une technique standard permettant d'évaluer le système tel qu'il est distribué.

Macr` -étal` ns

Contest est un exemple type de macro-étalon de performance de systèmes d'exploitation. Il est dédié à l'évaluation de la performance du noyau *Linux*. Il s'agit de déterminer le temps nécessaire à la compilation d'un noyau *Linux* en présence de différentes activités. Le rôle de ces activités est d'enclencher les mécanismes du noyau comme la gestion des processus, de la mémoire ou des entrées/sorties.

Cinq mesures sont associées à chaque exécution de l'étalon :

- Le temps nécessaire pour la compilation du noyau,
- Le pourcentage moyen d'utilisation CPU pour la compilation,
- Le temps nécessaire à l'exécution des activités,
- Le pourcentage moyen d'utilisation CPU pendant l'exécution des activités,
- Le rapport du temps de compilation obtenu en présence d'activités sur le temps obtenu en absence d'activités. Les mesures concernent donc le système dans sa globalité.

Lors de la comparaison de la performance de plusieurs noyaux, on recherchera celui auquel correspond un rapport de temps de compilation minimum et un pourcentage élevé d'utilisation

CPU, synonyme d'une utilisation optimale des ressources matérielles. Pour des raisons d'impartialité, la comparaison entre plusieurs versions du noyau doit se faire sur le même matériel, avec la même version du noyau à compiler, la même version du compilateur et sans modifier le réglage du matériel. L'exécution de l'étalon sur un matériel différent reste toutefois intéressante pour comparer la mise en œuvre d'une même version du noyau sur différentes plates-formes.

La principale différence entre les étalons de sûreté de fonctionnement et les macro-étalons de systèmes d'exploitation concerne l'ensemble des mesures. Dès lors, cela explique que, lors de leurs conceptions, les premiers se sont inspirés des seconds.

Micr` -étal` ns

Les micro-étalons sont plus précis que les macro-étalons. Ils visent l'évaluation de la performance d'un service en particulier, comme la latence d'un appel système, le temps de changement de contexte et la performance du système de fichiers [Ousterhout 1990]. Ils ont notamment permis de mettre en évidence le fossé qui se creuse entre l'évolution des performances du matériel par rapport à celles du système d'exploitation.

Unixbench [Smith 1995] est un micro-étalon de performance dédié à l'évaluation de la performance des systèmes *Unix*. La performance de divers mécanismes présents dans la plupart des systèmes *Unix* est déterminée grâce à un ensemble d'activités. Chaque activité est dédiée à la mesure de la performance d'un mécanisme donné. Nous verrons par la suite que sa capacité à activer les mécanismes du système peut être très utile dans le cadre d'un étalon de sûreté de fonctionnement.

Lmbench a été introduit dans [McVoy & Staelin 1996]. L'objectif de cet outil est d'identifier et d'évaluer les goulots d'étranglement de la performance d'un système d'exploitation. C'est un ensemble de programmes qui mesure la latence et la bande passante de certains mécanismes internes d'un système d'exploitation comme la bande passante des opérations d'entrées/sorties ou le temps de changement de contexte. *HbenchOS* [Brown & Seltzer 1997] est une évolution de *Lmbench* pour augmenter sa flexibilité, sa précision et pour le rendre plus reproductible.

Remarques

Ces programmes peuvent êtres utilisés à la fois par les utilisateurs et par les concepteurs de systèmes. Ils permettent la comparaison de performances d'un même système sur des architectures différentes, différentes solutions logicielles pour un même système ou enfin la comparaison de différents systèmes. La plupart du temps, il existe une version pour divers systèmes *Unix*. De plus, certains disposent désormais d'une version sur les produits commerciaux de *Microsoft* et d'*Appl*e. Les résultats obtenus mettent en évidence l'importance critique de la performance des opérations associées à la mémoire sur la performance globale.

Les micro-étalons ont inspiré des études de sûreté de fonctionnement détaillées de certains services du noyau. Nous les aborderons à la suite de ce chapitre.

2.2 Évaluati` n de la sûreté de f` ncti` nnement

Pour évaluer la sûreté de fonctionnement d'un système, deux classes de techniques d'évaluation existent :

- les méthodes basées sur des modèles,
- les méthodes basées sur des mesures expérimentales.

Parmi les méthodes basées sur les modèles, la modélisation analytique se situe à un très haut niveau d'abstraction. Elle convient donc parfaitement à l'étalonnage de système car on ne dispose pas forcément d'informations détaillées sur celui-ci. Elle sert généralement de support pour la sélection d'une architecture sûre de fonctionnement lors de la conception d'un système informatique.

2.2.1 Méthode analytique

La modélisation analytique se base sur la description du comportement du système en prenant en considération les défaillances et la maintenance des composants (matériels et logiciels) du système ainsi que leurs interactions. Les mesures de sûreté de fonctionnement sont évaluées en allouant des valeurs aux paramètres des processus stochastiques du modèle. Les modèles analytiques sont reconnus comme un facteur rationnel de prise de décision entre plusieurs solutions architecturales, incluant les politiques de maintenance, pendant la phase de conception d'un système tolérant aux fautes [Jarboui *et al*. 2002].

La modélisation nécessite de connaître :

- La composition du système et des interactions entre ses composants,
- Les mécanismes de détection d'erreurs et de tolérance aux fautes
- Les politiques de maintenance.

Les modèles les plus couramment utilisés sont basés sur les machines à états. Par rapport aux diagrammes de fiabilité et aux arbres de fautes [Laprie *et al*. 1995], ils ont l'avantage de permettre la prise en compte des dépendances stochastiques entre les composants de façon plus simple. Les chaînes de Markov constituent le type de modèle le plus utilisé pour modéliser la sûreté de fonctionnement d'un système informatique. En associant une structure de récompense au même modèle, il est possible de coupler des mesures de sûreté de fonctionnement à des mesures de performance.

La modélisation analytique peut être divisée en trois phases étroitement liées :

- Le choix des mesures de sûreté de fonctionnement à évaluer,
- La construction du modèle de description du comportement du système,
- Le traitement du modèle pour évaluer les mesures de sûreté de fonctionnement.

Cependant, et en dépit des avantages présentés par la modélisation par les chaînes de Markov, la modélisation d'un système complexe aboutit à une explosion du nombre d'états. En effet, la modélisation de la sûreté de fonctionnement de systèmes constitués de nombreux composants nécessite la description des comportements des éléments du système et de leurs interactions. L'utilisation de moyens d'aide à la construction des modèles est alors nécessaire. C'est pourquoi, pour faciliter la génération des modèles à grand nombre d'états, les langages de

spécification de haut niveau comme les réseaux de Petri stochastiques généralisés sont souvent utilisés.

Les réseaux de Petri ont été intégrés à de nombreux outils logiciels d'évaluation. On peut citer SPNP [Ciardo *et al.* 1989], SURF2 [Béounes *et al.* 1993], UltraSAN [Sanders *et al.* 1995] et DEEM [Bondavalli *et al.* 2000]. Certains intègrent des moyens de validation syntaxique et sémantique du modèle [Béounes *et al.* 1993]. Les événements, comme les défaillances ou les restaurations, sont représentés par leurs taux d'occurrence. La couverture de la détection d'erreurs, le recouvrement et l'efficacité des mécanismes de maintenance sont représentés par des probabilités conditionnelles. Ils permettent une représentation compacte du comportement du système et ils fournissent aussi des moyens de vérification structurelle du modèle.

Les applications des méthodes par composition de modèles sont multiples et variées. La méthode de modélisation par blocs présentée dans [Borrel 1996], par exemple, a été appliquée à la modélisation d'un sous-ensemble du système de contrôle du trafic aérien français [Kanoun *et al.* 1999]. Nous citons également la méthode par composition modulaire [Rabah & Kanoun 1999] qui a permis l'évaluation de la sûreté de fonctionnement d'un système multiprocesseurs à mémoire distribuée et partagée.

2.2.2 Techniques d'injection de fautes

L'injection de fautes s'inscrit dans le cadre à la fois de l'élimination et de la prévision des fautes. C'est un moyen privilégié de validation expérimentale des mécanismes de tolérance aux fautes. Les méthodes basées sur des mesures expérimentales permettent de valider et d'établir un lien entre le modèle et la réalité [Arlat *et al.* 1990]. Elles concernent les observations en vie opérationnelle du système et les expériences contrôlées d'injection de fautes dont le but est de caractériser le comportement du système en présence de fautes.

Dans un cadre de prévision de fautes, l'injection de fautes vise à reproduire des conditions proches de la réalité. Elle possède alors un double objectif :

- Comprendre le comportement du système en présence de fautes.
- Prévoir le comportement erroné du système et estimer l'efficacité des mécanismes de tolérance aux fautes en termes de couverture et de latence.

L'injection de fautes est couramment utilisée pour caractériser les modes de défaillance des COTS. Cette caractérisation permet d'élaborer des solutions architecturales comme les mécanismes de confinement d'erreurs afin de mieux détecter et tolérer leurs erreurs [Salles *et al.* 1999].

L'injection de fautes consiste à insérer des fautes de façon artificielle dans un système informatique ou dans un modèle de simulation pour comprendre l'effet des fautes réelles et de fournir les retours nécessaires pour l'amélioration du système. Les études d'injection de fautes destinées à valider un système tolérant les fautes se caractérisent par deux critères principaux :

- Le niveau d'abstraction du système cible,
- La forme d'application de l'injection de fautes.

Les méthodes d'injection de fautes sont multiples, variant de la simulation de fautes sur un modèle du système (ex. *Mefisto-L* [Arlat *et al.* 1999]) à l'injection physique sur un prototype (ex. *Messaline* [Arlat *et al.* 1990]). La tendance actuelle s'oriente vers l'injection de fautes par logiciel, grâce à sa flexibilité et à son coût peu élevé. Étant facile à appliquer, l'injection de fautes par logiciel cible actuellement toutes les couches d'un système informatique et spécialement les couches logicielles : le logiciel exécutif dans [Kao *et al.* 1993] et [Rodríguez *et al.* 1999], les intergiciels dans [Marsden & Fabre 2001] et [Pan *et al.* 2001] ainsi que les applications [Tsai & Singh 2000].

En dépit de leur flexibilité, les techniques d'injection de fautes par logiciel présentent un certain nombre d'inconvénients qui sont résumés ci-dessous :

- Il n'est pas toujours possible d'injecter des fautes par logiciel dans des endroits particulièrement protégés. Par exemple certains registres du processeur sont très difficiles d'accès.
- L'instrumentation du logiciel perturbe l'exécution de l'activité sur le système cible et peut même changer sa structure originelle. Une conception soignée peut minimiser les effets de cette intrusivité.
- La faible résolution du temps d'exécution des logiciels engendre des problèmes de fidélité par rapport aux résultats fournis. Ce problème n'est pas perceptible dans le cas de fautes de grande latence comme les fautes dans la mémoire. Cependant, dans le cas des fautes de faible latence comme les fautes dans les processeurs, nous sommes dans l'incapacité d'observer certains états erronés.

Un aspect très important concerne les conséquences des fautes injectées : les erreurs. En pratique, une même erreur peut être provoquée par des fautes différentes. Cette constatation a été le point de départ du développement des techniques d'injection de fautes par logiciel. Ainsi, l'inversion des bits dans les registres du processeur ou dans la mémoire permet de simuler des fautes matérielles transitoires et en partie certaines fautes logicielles [Costa *et al.* 2001]. Plusieurs études ont montré que les erreurs provoquées par ces simples inversions de bit étaient similaires à celles qui sont provoquées par des injections physiques [Fuchs 1996].

L'objectif des expériences d'injection de fautes étant la caractérisation du comportement du système en présence de fautes, elle semble donc particulièrement intéressante dans le cadre de la problématique soulevée au premier chapitre.

2.3 Étal` n de sûreté de f` ncti` nnement p` ur les systèmes d'expl` itati` n

Depuis plusieurs années, les étalons de performance permettent une comparaison objective de systèmes informatiques du point de vue de la performance. Comme présentées dans le paragraphe précédent, les mesures fournies sont de types mesures temporelles et utilisation du microprocesseur. Des consortiums et des accords ont été développés pour normaliser ces étalons, afin de standardiser leurs mesures de performances. Cette reconnaissance de la part de la communauté est essentielle pour la crédibilité de la mesure. Les concepteurs de systèmes et

les utilisateurs sont ainsi à même de choisir les composants et les systèmes en fonction de critères reconnus et de mesures comparables.

Les étalons de sûreté de fonctionnement actuels prennent leurs origines dans les techniques d'évaluation de la sûreté qui sont utilisées lors de la phase de validation des systèmes informatiques. Ils s'inspirent également des principes des étalons de performance. En effet, une part des problèmes posés a été résolue par des techniques du domaine des performances.

Cependant, il n'existe pas d'étalon de sûreté de fonctionnement standard pour les systèmes d'exploitation. Les concepteurs de systèmes cherchent aujourd'hui de plus en plus à évaluer les composants intégrables à leur système, en particuliers vis-à-vis de leur comportement en présence de fautes. C'est dans cet objectif que des recherches ont été effectuées sur les étalons de sûreté de fonctionnement. Parmi elles, on relèvera les tests de robustesse [Koopman *et al.* 1997], [Mukherjee & Siewiorek 1997] et les travaux qui ont unifié l'étude de la performance et de la sûreté de fonctionnement [Tsai *et al.* 1996]. Il existe diverses techniques d'évaluation de la sûreté de fonctionnement, ciblant différents niveaux d'un système informatique. Toutefois, aucune approche standard permettant la comparaison globale de systèmes ou de composants n'a été acceptée par la communauté.

Pour pallier cette lacune, nous définissons successivement les dimensions d'un étalon de sûreté de fonctionnement de systèmes d'exploitation. Elles sont illustrées dans la Figure 2-1 [Kanoun *et al.* 2002].

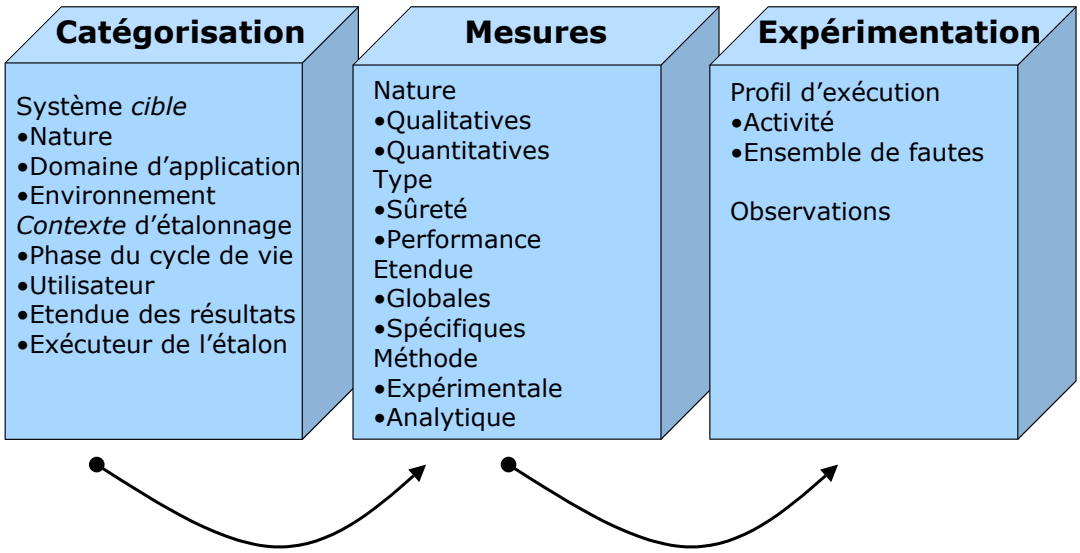


Figure 2-1 Dimensions d'un étalon de sûreté de fonctionnement

2.3.1 Dimensions de catégorisation

Avant de présenter les mesures requises et les dimensions expérimentales d'un étalon de sûreté de fonctionnement des systèmes d'exploitation, on doit définir les frontières du système cible et son contexte d'étalonnage. Les dimensions de catégorisation permettent d'organiser cet espace et l'étape préliminaire de conception de l'étalon se limite à leur définition.

Les systèmes cibles sont généralement les systèmes d'exploitation à usage général tels *Windows* ou *Linux*. Il serait appréciable de décrire le système en détail pour pouvoir évaluer des mesures spécifiques comme, par exemple, celles qui sont directement liées aux mécanismes de tolérance aux fautes disponibles sur le système. Mais le recours à une description détaillée du système peut réduire les chances de portage de l'étalon à d'autres systèmes. Dans le paragraphe précédent, nous avons mis en évidence la diversité des systèmes d'exploitation. Il est cependant possible de généraliser leur fonctionnement en considérant un système d'exploitation comme une couche logicielle qui fournit un certain nombre de services.

La Figure 2-2 illustre une décomposition d'un système informatique basé sur un système d'exploitation. Le noyau du système d'exploitation est la base du bon déroulement de l'exécution des applications. C'est l'élément critique du système et donc le composant ciblé. Il assure le lien entre les applications et le matériel en se basant notamment sur les logiciels de pilotage des périphériques. Les autres éléments sont considérés à l'extérieur du système cible, et définissent l'environnement opérationnel du noyau.

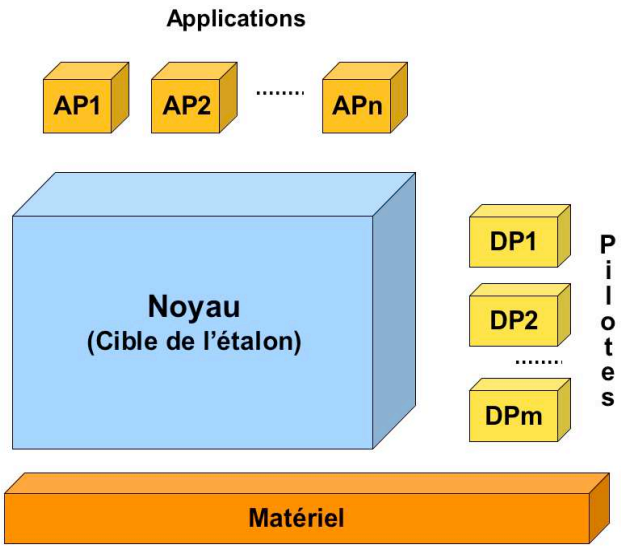


Figure 2-2 Catégorisation des systèmes d'exploitation

Nous considérons que l'exécuteur de l'étalon est différent du développeur du système d'exploitation cible. C'est pourquoi nous supposons une connaissance limitée de la structure interne du système. Cependant, les interfaces d'un noyau sont connues et documentées. Nous reviendrons plus en détails sur ce point au cours du chapitre trois.

2.3.2 Mesures

L'objectif majeur d'un étalon est d'évaluer un composant en termes de caractéristiques et de mesures. Le but de cette caractérisation est la possibilité de comparer plusieurs systèmes, comme, par exemple, des candidats à une intégration dans un système plus important.

La sûreté de fonctionnement couvre un large spectre de mesures [Laprie *et al.* 1995]. Parmi elles, nous pouvons citer la disponibilité, la fiabilité et la sécurité innocuité. Les mesures à

évaluer dépendent du domaine d'application du système cible. Globalement, son comportement peut être perçu par ses utilisateurs comme une alternance entre deux états :

- Service correct, où le service attendu est délivré,
- Service incorrect, où le service attendu n'est pas délivré.

Une défaillance est définie par la transition entre l'état service correct et l'état service incorrect du système. Inversement une restauration est définie par la transition entre l'état service incorrect et l'état service correct du système. Les mesures fournies par un étalon de sûreté de fonctionnement dépendent des dimensions de catégorisation et peuvent être qualitatives ou quantitatives :

- Les mesures qualitatives correspondent à des propriétés du système relatives à la sûreté de fonctionnement comme la présence de codes détecteurs d'erreurs dans la mémoire permettant, par exemple, d'agir pour éviter une défaillance.
- Les mesures quantitatives correspondent à une évaluation des attributs de la sûreté de fonctionnement du système. Elles se basent principalement sur l'analyse des défaillances du système.

L'activation d'une faute sous la forme d'une erreur peut entraîner soit la défaillance totale du système, soit un comportement dégradé dû à une diminution des performances. Les mesures de performance et de sûreté de fonctionnement sont donc très étroitement liées. C'est pourquoi l'évaluation de la performance d'un système en présence de fautes permet de caractériser en partie le comportement du système du point de vue de la sûreté de fonctionnement.

L'environnement opérationnel d'un système d'exploitation peut se résumer aux applications, au matériel et aux logiciels de pilotage des périphériques. La robustesse d'un système d'exploitation est sa capacité à résister aux fautes pouvant être introduites par son environnement opérationnel. Elle est mesurée grâce à l'observation de son comportement en présence de fautes externes, c'est-à-dire des fautes provenant de son environnement. La méthodologie présentée dans le chapitre trois de ce manuscrit a pour objectif d'évaluer la robustesse du noyau d'un système d'exploitation.

Les mesures fournies par les étalons de performance permettent la comparaison de la performance de certains mécanismes et services fournis par les systèmes d'exploitation. Aucun de ces étalons ne fournit de mesures de sûreté de fonctionnement. Cependant, il est possible de les réutiliser dans un contexte d'étalonnage de sûreté de fonctionnement pour pouvoir :

- Sensibiliser les mécanismes de base du système d'exploitation considéré pendant le relevé des mesures de sûreté de fonctionnement.
- Mesurer la « dégradation » de la performance en présence de fautes.

Nous présentons dans la suite de ce paragraphe les différentes mesures qui permettent de caractériser la sûreté de fonctionnement d'un système d'exploitation donné. En plus des mesures de robustesse, nous citerons quelques mesures temporelles et les mesures relatives à la capacité du système cible à gérer la propagation d'erreurs entre processus.

Mesures de r` bustesse

Comme nous l'avons exprimé en introduction de ce paragraphe, la robustesse est une quantification de la capacité d'un système ou d'un composant à fonctionner correctement en présence d'entrées invalides ou de conditions environnementales stressantes. Nous distinguons les mesures de robustesse au niveau du système d'exploitation de celles qui sont relatives au comportement des applications. De plus, nous définissons deux classes d'observations effectuées :

- Les défaillances non rapportées : arrêt, blocage ou résultat incorrect du processus. Nous les nommerons « modes de défaillances ».
- Les défaillances rapportées : retour de codes d'erreurs et traitement d'exceptions. Nous leurs ferons référence sous la dénomination « notifications d'erreurs ».

Une mesure intimement liée à la mesure de robustesse a trait à la pertinence des mécanismes de détection d'erreurs. Le traitement des conditions exceptionnelles dépend de l'exactitude de ces mécanismes. L'analyse du code d'erreur retourné par le noyau doit être comparée à la valeur qu'il devrait avoir en présence de la faute, nommée « oracle », ce qui rend l'exercice difficile.

Mesures temp` relles

Les mesures temporelles, comme les temps de redémarrage et d'exécution, sont très appréciées par les intégrateurs de systèmes et elles sont capitales dans le cas d'applications « temps réel », où le temps est un facteur déterminant. Le temps de redémarrage est généralement déterministe, mais sa valeur est susceptible d'évoluer dans le cas d'un état défaillant du système. Sa valeur dépend de la durée des vérifications et des corrections nécessaires pour restaurer l'intégrité du système. Ce type de mesures est exclusivement obtenu grâce aux étalons de sûreté de fonctionnement. Les mécanismes internes de gestion des processus dans un système d'exploitation sont également une cible privilégiée des mesures temporelles en présence de fautes. Ces mesures temporelles, comme le temps nécessaire au changement de processus, dépendent significativement de l'état du système.

L'obtention de certaines mesures temporelles nécessitent une connaissance détaillée de l'architecture du noyau du système d'exploitation [Rodriguez *et al.* 1999]. Elles sont donc un frein à la portabilité de l'étalon en ne respectant pas le concept de vision « boîte noire ». Par contre, d'autres mesures comme le temps minimum d'exécution d'un appel système, réalisant le basculement des privilèges d'exécution (utilisateur à noyau), sont facilement évaluables. De telles mesures sont utilisées pour des systèmes d'exploitation à capacité réduite dans le cadre de systèmes embarqués.

Le temps nécessaire pour retourner un code d'erreur représente une autre mesure temporelle qui associe une mesure de performance à une caractéristique de sûreté de fonctionnement, la détection de l'erreur.

Ces différentes mesures ont pour objectif de caractériser le comportement du système d'exploitation. La comparaison entre deux systèmes d'exploitation consiste à étalonner chacun d'eux et à comparer les mesures obtenues. Cependant, si l'application est disponible, il est

possible d'étalonner l'ensemble système d'exploitation et application. Ainsi les mesures fournies par l'étalon caractérisent le couple formé des deux logiciels.

2.3.3 Dimensions d'expérimentation

Ces dimensions incluent tous les aspects relatifs à l'expérimentation sur le système cible pour obtenir les mesures sélectionnées. Elles comprennent l'activité du système, l'ensemble de fautes représentées et les observations effectuées. Ces trois dimensions dépendent des dimensions de catégorisation et de mesures.

L'activité représente le profil opérationnel du domaine d'application considéré. Son rôle est de générer suffisamment d'opération pour simuler un état du système conforme à la réalité. Des étalons de performance largement reconnus fournissent généralement des activités suffisamment représentatives de plusieurs domaines d'application. Cependant, l'activité recherchée dans un contexte d'étalonnage de sûreté de fonctionnement doit prendre en compte l'impact des actions de maintenance préventives ou correctives qui peuvent faire partie de la politique de tolérance aux fautes associées au système comme les procédures de prévention et de recouvrement de fautes. Alors les étalons de performance ne suffisent plus à définir une activité réaliste.

L'ensemble de fautes inclut trois notions :

- Les fautes internes, dont le but est de simuler à la fois des fautes d'origine physique et d'éventuelles fautes de conception du noyau candidat.
- Les fautes externes, dont le but est de simuler des entrées erronées ou stressantes.
- Les conditions exceptionnelles, décrivant le cas où des valeurs peuvent appartenir au domaine de validité des entrées du système, mais pour lesquelles le système cible n'est pas obligatoirement protégé.

L'ensemble de fautes doit simuler les menaces réelles que le système est susceptible d'affronter par des classes de fautes. Le niveau de ces classes dépend du niveau de détail mis à la disposition du développeur de l'étalon. Pour un étalon, une des principales difficultés à l'obtention d'un standard reconnu par la communauté et le domaine est la représentativité des fautes étudiées. Vu l'importance de cette dimension, un effort est nécessaire pour avoir confiance dans les résultats retournés par les expériences d'injection de fautes logicielles.

Si le système cible est un logiciel, l'injection de fautes internes risque de modifier l'état du système initial. Toutefois, pour des systèmes cibles possédant un niveau de granularité assez élevé, il est possible d'étudier l'influence d'un module interne erroné sur le comportement des autres modules du système. Par exemple, les travaux présentés dans [Hiller *et al.* 2001] ont permis de sélectionner les meilleurs emplacements de mécanismes de détection d'erreurs grâce à l'injection de fautes internes dans le système. Si le développeur de l'étalon est différent du développeur du système, l'intégration de fautes internes dans l'ensemble de fautes reste complexe dans la pratique. En effet, il est difficile d'avoir accès aux fautes logicielles internes (voir le concept de « boîte noire ») des systèmes commerciaux propriétaires, car, pour des raisons commerciales, les développeurs de tels systèmes affirment l'absence de fautes dans leurs systèmes. A contrario, les fautes externes dépendent entièrement de l'environnement

opérationnel et incluent les conditions exceptionnelles. L'injection de fautes externes permet d'analyser les failles de robustesse du système et peut aboutir à l'élaboration de mécanismes de confinement d'erreurs pour améliorer la robustesse du système.

Les observations permettent de caractériser le comportement du système cible après l'exécution de l'activité, en présence de l'ensemble de fautes. Les mesures associées à l'étalon de sûreté de fonctionnement sont obtenues par le traitement de ces observations. Nous verrons dans le chapitre trois comment obtenir des mesures de sûreté de fonctionnement pertinentes. Les observations dépendent du niveau de détail du système cible et de sa capacité à fournir les mécanismes nécessaires à l'observation de son comportement. Certains systèmes dits réflexifs fournissent des mécanismes dédiés à la fois au contrôle et à l'observation du comportement du système. Les auteurs des travaux présentés dans [Salles *et al*. 1999] proposent la définition d'une méta-interface à travers laquelle il est possible de contrôler et d'observer le comportement du micronoyau *Chorus*.

2.4 Caractérisati` n des systèmes par injecti` n de fautes

Nous avons vu précédemment que l'évaluation de la sûreté de fonctionnement des systèmes d'exploitation en présence de fautes est liée à l'utilisation de composants COTS dans le développement des systèmes informatiques. Il est devenu essentiel de pouvoir évaluer la confiance, en termes de sûreté de fonctionnement, qu'il est possible d'accorder à ces composants. En effet, ces composants n'ont pas eu à respecter les contraintes imposées lors de la spécification du système censé les intégrer.

L'utilisation des COTS dans la construction des systèmes informatiques permet d'améliorer la souplesse de la phase de conception d'un système. Toutefois, les développeurs de systèmes sont confrontés à deux problèmes posés par cette nouvelle tendance :

- Les propriétés de sûreté de fonctionnement intrinsèque du composant candidat à l'intégration, c'est-à-dire la robustesse de son interface et la couverture de ses mécanismes de détection d'erreurs,
- Les conséquences que peut avoir sa défaillance sur le système final.

L'analyse du comportement du composant par injection de fautes est une manière de répondre à ces interrogations. Nous présentons, dans ce qui suit, quelques techniques relatives à la caractérisation de systèmes d'exploitation par injection de fautes. Nous présenterons dans un premier temps les techniques dont le but est de caractériser le comportement des systèmes d'exploitation en utilisant à la fois des modèles de fautes matérielles, simulant des fautes physiques, et des modèles de fautes logicielles, simulant des fautes logicielles internes. Dans un deuxième temps, nous présenterons deux techniques de caractérisation de la robustesse des systèmes d'exploitation par rapport aux fautes externes dans les applications. Nous clôturerons cette section par les travaux sur l'évaluation de l'impact des pilotes de périphériques et des autres extensions du noyau sur le système.

2.4.1 Caractérisation par rapport aux fautes internes et matérielles

Dans le cadre de la caractérisation de noyaux en présence de fautes, nous verrons des exemples de techniques d'injection de fautes par logiciel qui consistent à injecter des erreurs à l'intérieur de l'espace mémoire du noyau. Nous verrons les outils suivants :

- *FINE*,
- *MAFALDA*,
- *XCEPTION*.

Le but de ces travaux est de simuler à la fois des fautes d'origine physique et d'éventuelles fautes de conception du noyau candidat.

FINE

Parmi les travaux pionniers sur l'injection de fautes internes dans un système d'exploitation, nous notons particulièrement ceux rapportés dans [Kao *et al.* 1993]. L'objectif de ces travaux est d'étudier les canaux de propagation d'erreurs dans le noyau *Unix* et d'évaluer l'impact de divers types de fautes sur le comportement du noyau grâce à l'outil d'injection de fautes *FINE*. Les modèles de propagation d'erreurs sont construits pour les fautes matérielles et les fautes logicielles.

Seules les fautes matérielles qui ont une influence directe sur le flot d'exécution des programmes ont été considérées. Ces fautes sont classées par la localisation des manifestations dans : la mémoire, le processeur, le bus, les entrées/sorties. Ces fautes sont soit permanentes (collage à 0 ou à 1), soit transitoires (inversion de bits).

Les fautes logicielles sont issues des données opérationnelles collectées après l'analyse des causes de défaillance rapportées des systèmes logiciels entre autre dans [Tang & Iyer 1992] et [Lee & Iyer 1992]. Les fautes logicielles considérées dans le cas de *FINE* sont les fautes d'initialisation, d'affectation, de branchement ou affectant une fonction.

Les résultats expérimentaux ont montré que les fautes affectant la mémoire et les fautes logicielles ont généralement une latence de détection élevée. Les fautes affectant le bus et le processeur ont tendance à causer un arrêt immédiat du système.

MAFALDA

MAFALDA [Arlat *et al.* 1999], acronyme anglais de « *Microkernel Assessment by Fault injection AnaLysis and Design Aid* » est un outil développé au LAAS-CNRS. Ce prototype permet de caractériser le comportement d'un micronoyau en présence de fautes, de faciliter la mise en œuvre de mécanismes de détection d'erreurs. L'injection de fautes dans *MAFALDA* s'effectue dans les segments de mémoire du micronoyau.

Il s'agit d'inversions uniques de bit destinées principalement à :

- Simuler des fautes matérielles, lorsque l'injection vise un segment de données,
- Simuler des fautes logicielles internes lorsque l'injection vise un segment de code correspondant à des données.

Une activité spécifique est associée à chaque composant fonctionnel du micronoyau cible (par exemple : synchronisation, ordonnancement, etc.). *MAFALDA* procède à une distribution aléatoire et uniforme des fautes injectées, sur l'espace d'adressage du module cible, mais ne réalise que les expériences où les fautes sont effectivement activées.

En plus de la caractérisation des modes de défaillance du noyau, *MAFALDA* permet de déduire les canaux de propagation existants entre différents modules fonctionnels et de calculer la latence de détection des exceptions levées par le processeur. L'évaluation des canaux de propagation d'erreurs à l'intérieur du micronoyau *Chorus* a révélé les relations de dépendance fonctionnelle entre les modules. Cette étude expérimentale a également montré la différence des distributions des erreurs observées (mode de défaillance) après l'injection de fautes dans le segment de texte et le segment de données.

Les premiers résultats ont ciblé la caractérisation du micronoyau *Chorus* [Systems 1997]. Par la suite, l'outil a également été utilisé pour analyser *LynxOS* [LynuxWorks]. De plus, une version améliorée, nommée *MAFALDA-RT*, a vu le jour. Cette version inclut des aspects temps réel (temps de réponse, dépassement de limites temporelles, etc.) [Rodriguez *et al.* 2002]. Les concepts de base de nos travaux d'analyse présentés dans le chapitre quatre ont été également appliqués à ces résultats.

XCEPTION

L'outil d'injection de fautes *XCEPTION* [Carreira *et al.* 1998] a été utilisé pour évaluer le comportement du noyau temps-réel *LynxOS* en présence de fautes. Les fautes injectées consistent en une inversion de bit dans le processeur ou dans la mémoire, pendant l'exécution du code du noyau ou d'une application, pour simuler les effets d'une corruption isolée[10]. Les fautes suivent une distribution uniforme dans le temps. Deux types d'activités ont été utilisées : synthétiques et réelles.

Les fautes injectées pendant l'exécution du code du noyau causent l'arrêt du noyau ou n'aboutissent à aucun résultat visible. Les résultats issus des expériences d'injection de fautes ont montré aussi que les applications qui utilisent rarement les appels système produisent plus de résultats erronés que celles qui utilisent intensivement les services du noyau. De plus ces expériences ont montré que *LynxOS* est assez robuste puisque la majorité des fautes qui ont provoqué des paramètres erronés des appels système ont été détectées par le noyau. Par ailleurs le faible pourcentage des erreurs qui n'ont pas été détectées plaide pour le développement de mécanismes de détection d'erreurs au niveau de l'API de *LynxO*S. Les derniers travaux concernant *XCEPTION* sont rapportés dans [Madeira *et al.* 2002b] et concernent l'évaluation expérimentale d'un système embarqué pour les applications spatiales à base de COTS.

2.4.2 Caractérisation par rapport aux fautes externes

Nous avons vu dans le chapitre précédent que les fautes internes et matérielles n'étaient pas la principale source d'erreurs des systèmes d'exploitation. De plus, l'objectif de nos travaux vise entre autre à la caractérisation de COTS en vue d'une intégration. La caractérisation des COTS

[10] *Single Event Upset*

vis-à-vis des fautes externes est donc un problème majeur. Elle se définit par le terme « robustesse », présenté en début de ce chapitre. Le but des tests de robustesse est de caractériser le comportement du système en présence d'entrées erronées ou stressantes. La recherche dans le domaine des tests de robustesse a commencé à prendre de l'ampleur au début des années 1990. Parmi les premiers travaux, ceux rapportés dans [Siewiorek *et al.* 1993] et [Mukherjee & Siewiorek 1997] définissent le test modulaire. Les tests sont construits en considérant le système comme une collection de modules isolés. Plusieurs études ont tenté d'évaluer la robustesse des systèmes logiciels. Dans [Miller *et al.* 2000] les auteurs ont examiné le comportement d'utilitaires *Unix* et *Windows* en présence d'entrées générées aléatoirement. Crashme [Carrette 1996] est un autre exemple de test de robustesse des systèmes *Unix* et *Windows*. Ce programme remplit un tableau avec des données aléatoires et l'exécute comme si c'était du code. Enfin, on peut citer deux exemples de travaux présentés dans [Koopman *et al.* 1997] et [Rodríguez *et al.* 1999] qui ont abouti à la génération de deux outils qui supportent la conduite des expériences de tests de robustesse. Le premier, *Ballista*, caractérise l'efficacité des mécanismes de gestion des exceptions des modules logiciels. Le second, *MAFALDA*, a déjà été introduit dans le paragraphe précédent.

BALLISTA

BALLISTA a été développé à l'université de Carnegie-Mellon à Pittsburgh. L'outil teste la robustesse de COTS par rapport à des entrées exceptionnelles mais valides [Koopman & DeVale 1999].

Cette approche a été appliquée dans un premier temps aux systèmes d'exploitation supportant une interface POSIX, puis elle a été portée aux systèmes de la famille *Windows* ciblant ainsi l'interface standard WIN32 de *Microsoft* [Shelton *et al.* 2000]. Les tests se composent d'une combinaison de valeurs exceptionnelles des paramètres des appels systèmes. Chaque appel est effectué une seule fois par test avec un ensemble particulier de paramètres. À chaque paramètre est associé un certain nombre de valeurs prédéfinies exceptionnelles mais valides. L'avantage majeur de cette approche est la génération automatique du code source des cas de tests avant exécution. Un test est un programme contenant la déclaration des paramètres et l'appel de la fonction. La génération automatique des combinaisons de paramètres possibles d'une fonction permet la détection des erreurs de conception logicielles. L'efficacité des mécanismes de détection d'erreurs peut également être mesuré en analysant la propagation de l'erreur.

La robustesse du système cible est analysée par rapport à une échelle, dénommée avec humour C.R.A.S.H, de cinq modes de défaillance :

- ***C****rash* : nécessite le redémarrage du système.
- ***R****estart* : redémarrage de la tâche.
- ***A****bort* : interruption ou arrêt anormal de la tâche.
- ***S****ilent* : pas de code d'erreur retourné.
- ***H****indering* : le code d'erreur retourné n'est pas le bon.

Les trois premiers modes de défaillance sont observés automatiquement et les deux derniers nécessitent un traitement particulier difficile à automatiser.

Les résultats obtenus en utilisant *BALLISTA* pour 15 systèmes d'exploitation POSIX montrent la présence du mode de défaillance Catastrophique dans six systèmes d'exploitation. Le redémarrage d'application n'a été observé que dans deux cas. Le mode de défaillance Arrêt d'exécution quant à lui a été révélé par tous les systèmes d'exploitation. Par ailleurs, le mode de défaillance Silencieux ne garantit pas forcément un comportement sûr de fonctionnement. C'est le mode de défaillance le plus difficile à détecter. Une comparaison multi-version des résultats obtenus a permis d'identifier ces modes de défaillance.

À l'origine, *BALLISTA* était basé sur une approche de test en boîte noire au niveau de l'API du noyau. Cette étude a permis la comparaison entre *Linux* et un ensemble de systèmes d'exploitation de la famille *Windows*. Globalement, les résultats sont similaires entre *Linux* et les dernières versions de *Windows*. Cependant, les résultats présentés dans [Shelton *et al*. 2000] ont été détaillés suivant les principales fonctionnalités d'un système d'exploitation. Une analyse fine montre que *Linux* possède une mise en œuvre des appels système plus robuste que celle de *Windows*, par contre *Linux* est plus sensible au mode « *Abort* » dans le cas des fonctions de la bibliothèque C.

Le travail le plus récent relatif à *BALLISTA* concerne son application à l'étude de la robustesse de plusieurs intergiciels basés sur une architecture de type CORBA[11] [Pan *et al*. 2001]. Ces travaux montrent la portabilité de l'outil et sa facilité d'utilisation à différents niveaux du système.

MAFALDA

Comme nous l'avons présenté dans la section précédente, *MAFALDA* permet l'injection de fautes dans les segments de mémoire du micronoyau. Mais l'outil permet également d'effectuer des corruptions au niveau des paramètres des appels système.

Les résultats obtenus en utilisant la technique d'injection *MAFALDA* sur le noyau Chorus révèlent qu'une bonne part des erreurs introduites dans le composant fonctionnel de communication est détectée par les mécanismes de détection d'erreurs du micronoyau. Le noyau Chorus bloque rarement en présence de fautes. Cependant, les défaillances applicatives relevées sont fréquentes et de nombreux cas (36%) n'ont pas eu d'effets relevés malgré l'activation des fautes. Par ailleurs, les résultats concernant le composant fonctionnel de synchronisation (SYN) révèlent des différences sensibles : les défaillances applicatives observées sont très fréquentes et constituent le seul événement observable relevé. Le fait que les mécanismes de synchronisation mis en œuvre par Chorus ne vérifient pas les paramètres d'entrée explique ces résultats. C'est pourquoi des mécanismes de confinement d'erreurs ont été développés spécifiquement pour le composant fonctionnel de synchronisation. Des expériences d'injection de fautes ont été menées après l'insertion de ces mécanismes de confinement dans le micronoyau et ont montré l'efficacité de ces mécanismes. En effet, aucune défaillance d'application n'a été observée en présence de ces mécanismes lors de l'injection de fautes dans les paramètres des primitives de synchronisation.

---

[11] *Commun Object Request Broker Architecture*

2.4.3 Caractérisation de systèmes vis-à-vis des extensions

Nous venons de voir dans les paragraphes précédents différents travaux de caractérisation des systèmes d'exploitation. Ils se rapportent à l'évaluation de l'impact des fautes matérielles, logicielles, internes et externes. Les travaux traitant des fautes externes sont proches des problèmes de pilotes de périphériques. En effet, lorsque le composant cible de notre évaluation de COTS est le noyau du système, les pilotes peuvent être considérés comme externes au composant.

À notre connaissance, très peu d'études ont été effectuées visant à analyser l'impact des défaillances des pilotes de périphériques sur le système d'exploitation :

- La méthode d'analyse d'état,
- Le contrôle du flot d'exécution du processeur,
- L'injection de fautes dans le code binaire des pilotes,
- L'approche analytique du problème des pilotes de périphérique.

Nous allons succinctement présenter ces travaux dans la suite de ce paragraphe.

Analyses d'état

La méthode "state analysis" a été élaborée par la société *Intel*. L'objectif de ces recherches est d'améliorer la robustesse des pilotes de périphériques vis à vis des défaillances matérielles.

Les auteurs présentent dans [Edwards & Matassa 2002] une méthodologie pour la conception d'un prototype d'injection de fautes pour valider des pilotes robustes. Un pilote robuste est un pilote pour lequel ses concepteurs ont prévu des mécanismes de tolérance vis-à-vis des fautes provenant de son environnement comme, par exemple ici, le matériel. Les auteurs ont eu une approche « boîte noire » concernant le code des pilotes. L'injection de fautes s'effectue donc dans la partie basse du système, celle concernant les abstractions matérielles. Des « crochets » ont été développés pour remplacer les accès mémoires traditionnels du système. Il s'agit de réécrire les macros du système en macros configurables pour une injection. Lors de l'exécution, les pilotes utilisent alors ces crochets pour accéder au matériel : l'injection consiste à corrompre, au niveau de ces crochets, les valeurs retournées, modifiant les valeurs d'entrée de la partie basse du pilote. Il s'agit donc d'évaluer la robustesse du pilote en simulant une faute externe aux pilotes.

La Figure 2-3 montre le prototype à l'étude et présente ses trois composants principaux :

- Le pilote de système, composant à l'essai, utilisant les crochets (CDI).
- Le moteur d'injection de fautes, corrompant les paramètres dans les crochets.
- L'interface commune pilote-matériel, composée de l'ensemble des crochets, réalisant l'abstraction matérielle. Les échanges entre les composants s'effectuent via des commandes IOCTL (*Input Output Control*).

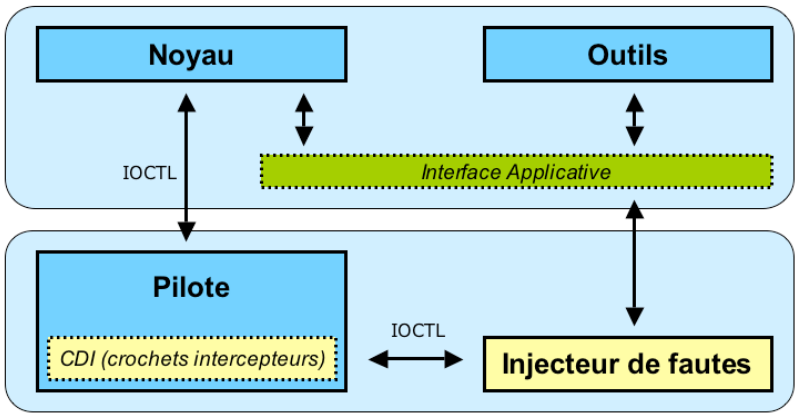


Figure 2-3 Prototype de la technique d'analyse d'état

Le prototype de contrôle des injections de fautes a montré qu'il est possible de créer un langage « machine d'état » et qu'il est possible d'observer l'état du matériel pendant les cycles normaux d'exécution d'un pilote. Les observations se divisent en deux catégories :

- Les alertes préviennent d'un accès inapproprié à un registre non disponible n'entraînant pas de conséquences graves sur le système.
- Les violations correspondent à une réaction non adaptée pouvant potentiellement provoquer une défaillance du système.

Lors d'une injection de faute, la surveillance des changements d'état du matériel aide à prendre des décisions. La technique est portable entre différentes versions d'un pilote et permet de comparer ses différentes évolutions lors des phases de test.

C` ntrôle du fl` t d'instructi` ns du pr` cesseur

Les travaux de l'université de l'Illinois [Gu *& al* 2003] présentent une étude expérimentale du noyau *Linux* en présence de fautes. L'objectif est de caractériser le comportement du système *Linux* vis-à-vis de fautes applicatives. L'article présente le problème des défaillances d'un système comme étant l'exécution d'instructions invalides dues, dans 95% des cas, à un déréférencement de pointeur[12], un défaut de page[13], un code opération invalide ou une faute de protection générale[14]. Pour caractériser le comportement du système en présences de fautes internes, il faut donc simuler ce genre d'erreurs et observer les réactions du système. La technique d'injection de fautes proposée est réalisée au niveau du flot d'exécution des instructions concernant le code du noyau.

[12] *Unable to handle kernel NULL pointer*

[13] *Unable to handle kernel page request*

[14] *Invalid opcode and general protection fault*

Les sous-systèmes visés sont :

- L'architecture (*arch*),
- Le système de fichiers virtuels (*fs*),
- Le cœur du noyau (*kernel*),
- Le gestionnaire de mémoire (*mm*).

La campagne d'injection comporte plus de 35.000 fautes affectant les fonctions, appartenant à ces sous-systèmes, les plus fréquemment utilisées dans l'étalon *UnixBench*, utilisé comme activité simulant un comportement en opération du système.

Trois types d'injection de fautes sont effectuées :

- Inversion d'un bit aléatoire dans une instruction,
- Inversion d'un bit aléatoire d'une instruction conditionnelle de branchement,
- Modification d'un bit pour inverser le sens d'une instruction de branchement conditionnelle.

Les Observations sont composées des mécanismes de détections d'erreurs du matériel et du système : les déréférencements de pointeurs, les défauts de pages, les codes opérations invalides, les paniques noyau, les problèmes de protection générale de la mémoire, les défauts de mémoire et des « *traps* »[15].

Les fonctions les plus sujettes à ce type d'injection ont été étudiées. Cette analyse montre qu'il est possible d'identifier des endroits stratégiques pour y inclure des mécanismes de tolérance aux fautes dans le code source d'un sous-système donné. L'objectif de ces mécanismes est de prévenir les erreurs provenant des applications et empêcher leur propagation dans le noyau. Cette technique peut être adaptée aux pilotes de périphériques en s'intéressant aux fonctions les plus fréquemment utilisées par les pilotes au lieu de celles utilisées par les applications.

Injecti` n de fautes dans le c` de binaire des pil` tes

Un des travaux les plus intéressants, car très proche de ceux présentés dans ce manuscrit, provient de l'université de Coimbra [Duraes & Madeira 2003]. Pour caractériser le comportement du système en présence de pilotes de périphériques défaillants, les auteurs présentent une technique d'injection de fautes pour introduire des fautes logicielles. Ces fautes simulent des fautes logicielles résiduelles dans les pilotes.

[15] *Divide error, init3, bounds, invalid TSS, overflow*

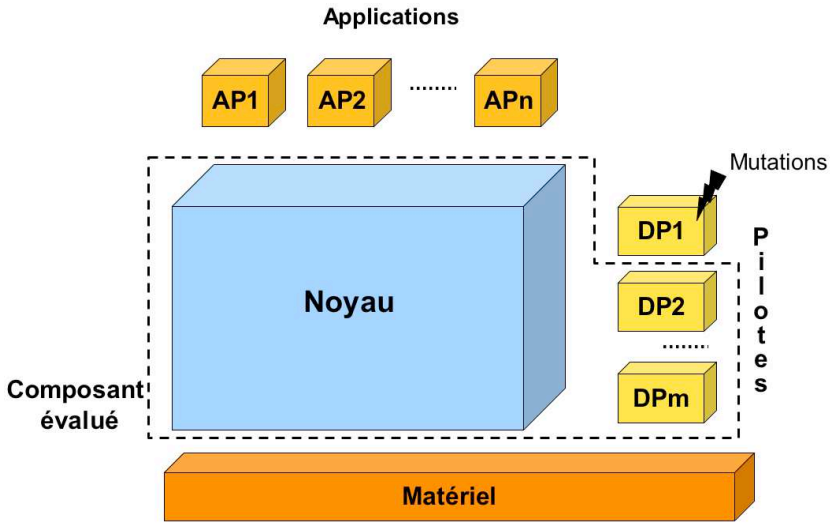


Figure 2-4 Insertion d'un pilote « mutant »

L'approche expérimentale concerne la famille *Windows*. Comme le montre la Figure 2-4, elle consiste plus précisément à corrompre un pilote de périphériques par une inversion de bits dans son code binaire, créant ainsi un code mutant du pilote. Celui-ci est alors inséré dans le système. Le système ciblé par l'outil correspond au noyau ainsi que les pilotes non corrompus. On observe alors le comportement du système lors de l'insertion et l'utilisation de ce pilote mutant. L'activité utilisée est spécifique au pilote et au système pour augmenter de manière intensive les interactions entre les deux.

L'observation du comportement du système en présence de fautes révèle alors des profils intéressants. Les modes de défaillances sont regroupés en douze catégories dont les critères sont :

- Le démarrage correct du système en présence du pilote défaillant,
- L'identification du pilote défaillant,
- Le redémarrage automatique de la machine,
- Les messages d'alertes du système,
- Le blocage du système,
- Les applications ont pu interagir avec le pilote,
- Le périphérique est désactivé,
- L'utilisateur est informé de la désactivation du système,
- Le périphérique est inutilisable,
- Pas d'observation,
- Pertes de données

Il est important de noter que de nombreuses observations sont relevées lors de la remise en opération du système (redémarrage). Les mesures sont alors fréquemment liées aux

vérifications effectuées au démarrage du système, limitant leur portée sur des systèmes critiques classiques (autre que les machines *Windows*). De plus, l'injection de fautes n'est pas effectuée pendant que le système est opérationnel. Elle est réalisée par un système tiers présent sur partition différente du disque dur de la machine cible avant le démarrage du système d'exploitation ciblé. Les fautes injectées enrichissent celle du modèle ODC [Chillarege *et al.* 1992] par la prise en compte de fautes logicielles de conception nouvelles : absence d'un appel à une fonction, absence ou corruption d'un code retour d'une opération du pilote et absence d'initialisation d'une variable locale.

Trois modèles de défaillances sont proposés pour caractériser le système :

- La disponibilité du système,
- Le retour d'expérience,
- La fiabilité des charges applicatives.

La technique d'injection de fautes utlisée dans cette méthode permet une grande portabilité de l'étalon. Toutefois, l'effet d'une injection de fautes par inversion de bit est difficile à analyser sur un système en opération classique.

Appr` che analytique du pr` blème des pil` tes de périphériques

La tendance actuelle dans l'évaluation de la sûreté de fonctionnement des systèmes d'exploitation est à l'injection de fautes. Cependant, [Chou *et al.* 2002] présente une approche radicalement différente appliquée sur *Linux*.

Il s'agit d'une étude des erreurs trouvées dans un système d'exploitation par l'analyse automatique et statique par compilation appliquée aux noyaux de *Linux* et OpenBSD. Trois métriques sont utilisées permettant d'obtenir :

- Les erreurs manuellement analysées,
- Les erreurs détectées par le vérificateur mais non-analysées manuellement,
- Le nombre d'occurrences de la vérification,
- Le nombre d'erreur, analysées ou non, divisé par les occurrences de ce type d'erreurs.

L'analyse statique est appliquée uniformément à l'ensemble du code source du système, en considérant une variété d'erreurs possible. L'automatisation permet d'obtenir des résultats sur de multiples versions et d'estimer le temps de correction des bogues. Les résultats révèlent que les modules de gestion de périphériques ont des taux d'erreur jusqu'à trois à sept fois supérieurs au reste du noyau. De plus, il semble que certaines fonctions et parties du système soient plus « riches » en erreurs que d'autres. Les deux vérificateurs fournissant le plus de résultats sont le *null checker* et le *block checker*. Le problème de ce genre d'étude est que le code source doit être disponible. Aujourd'hui, il n'y a guère que le noyau *Linux* qui est entièrement disponible parmi les noyaux de systèmes d'exploitation les plus répandus. Il en est de même pour les pilotes.

2.5 C` nclusi` n

Le but de l'étalonnage de sûreté de fonctionnement est de fournir des moyens génériques et reproductibles pour la caractérisation du comportement des systèmes en présence de fautes. Un étalon de sûreté de fonctionnement se distingue des techniques d'évaluation et de validation existantes par l'accord qu'il doit assurer entre les différentes communautés qui l'utilisent. Les mesures, les moyens de les obtenir et leur domaine de validité sont spécifiés à travers cet accord.

Dans un premier temps, nous avons présenté les généralités sur l'architecture et le fonctionnement des systèmes d'exploitation modernes. Nous avons vu ensuite un cadre générique qui permet de définir des étalons de sûreté de fonctionnement de ces systèmes, cadre inspiré des travaux sur les étalons de performances. Les systèmes concernés par ces techniques sont les logiciels COTS, comme les systèmes d'exploitation.

Le cadre d'étude de ces travaux se base sur 3 dimensions essentielles : la dimension de catégorisation, la dimension des mesures et la dimension d'expérimentation. Chacune de ces dimensions permet la réalisation d'un étalon de sûreté de fonctionnement reconnu, c'est-à-dire réussi. La mise en œuvre de ce cadre conceptuel s'appuie sur un ensemble de techniques allant de l'injection de fautes à la modélisation analytique et s'inspire également de la mise en œuvre pratique des étalons de performance. Ceci nous a amené à présenter l'état de l'art relatif à ces trois concepts :

- Les étalons de performance, en plus de l'état de l'art des étalons de sûreté de fonctionnement,
- Les étalons de sûreté de fonctionnement par modélisation analytique ou par injection de fautes.

Nous avons présenté à la fin de ce chapitre les différents travaux déjà effectués sur la sûreté de fonctionnement des systèmes d'exploitation vis-à-vis de fautes internes, externes et également concernant les pilotes de périphériques. À notre connaissance, très peu d'études ont été effectuées visant à analyser l'impact des défaillances des pilotes de périphériques sur le système d'exploitation. Les travaux d'évaluation sur *Linux* se concentrent plutôt sur la caractérisation de la robustesse des logiciels de pilotages de périphériques lorsqu'ils sont soumis à des défaillances du matériel. L'analyse complète de la fiabilité du noyau *Linux* ne distingue pas les pilotes et le cœur du noyau dans l'étude par injection de fautes, malgré leurs différences en fréquence d'erreurs. Dans l'approche par mutation du code, les auteurs ont simulé une défaillance d'un bit dans le code d'un pilote. Leur approche et la nôtre ont les mêmes objectifs en terme de sûreté de fonctionnement cependant leurs travaux s'intéressent de manière indifférente aux fautes logicielles et matérielles dans les pilotes. Une approche analytique proposée dans [Chou *et al.* 2001] a permis de mettre en évidence les nombreuses erreurs présentes dans les pilotes de périphériques *Linux*. Elle fut une des premières études évaluant l'impact des pilotes sur la sûreté de fonctionnement des systèmes d'exploitation.

Nous proposons dans ce manuscrit de compléter les travaux concernant l'étalonnage de la sûreté de fonctionnement des systèmes d'exploitation du point des pilotes de périphériques. L'approche doit permettre de définir une technique reconnue et portable pour évaluer l'impact des défaillances des pilotes de périphériques sur le noyau d'un système d'exploitation.

Chapitre 3 Méth` d` l` gie

Nous avons vu dans les chapitres précédents que plusieurs travaux d'étalonnage de sûreté de fonctionnement des systèmes d'exploitation ont été effectués vis-à-vis des défaillances des applications et du matériel. La problématique des pilotes a été, jusqu'ici, peu abordée. Pourtant c'est dans le code des pilotes que la plupart des bogues présents dans le système sont trouvés.

Dans ce chapitre, nous présentons une technique d'évaluation du noyau d'un système d'exploitation vis-à-vis de pilotes défaillants. Aussi, dans cet objectif, nous considérons que les extensions et le noyau sont deux composants différents du système d'exploitation. Bien qu'ils partagent de nombreuses caractéristiques en commun, nous pouvons les dissocier l'un de l'autre dans tous les systèmes. Comme nous allons le voir, l'étude des architectures des systèmes d'exploitation « grand public » les plus répandus, comme ceux des familles *Windows* et GNU/*Linux* présentées dans [Tsegaye & Foss 2004], ou ceux de la nouvelle génération des *MacOS*, met en évidence des similarités concernant les échanges entre les extensions et leur noyau. Les interactions entre un système d'exploitation et ses applications sont claires. L'API est une interface connue, stricte et bien définie dans tous les noyaux. En général, il n'en est pas de même pour l'interface entre le pilote et le noyau : l'interface est parfois confuse, les interactions entre les différents composants du noyau mal identifiées, les informations souvent partielles. C'est pour répondre à ce problème que nous avons cherché à identifier et décrire cette interface entre pilotes et noyau présente dans tous les systèmes d'exploitation actuels [Zaatar & Ouaiss 2002]. Nous précisons ensuite les critères de notre approche évoqués dans les chapitres précédents et les contraintes que nous nous sommes fixés pour remplir nos objectifs. Ils concernent principalement la portabilité de la méthode, les mesures adaptées à nos besoins et une souplesse de mise en œuvre. Nous décrivons enfin la méthode proposée pour évaluer le système vis-à-vis des défaillances des pilotes de périphériques en simulant une faute au niveau de l'interface pilote-noyau. Les principales étapes de cette approche : la technique d'injection, les profils d'exécution, les observations et les mesures.

3.1 Interface entre les pil` tes et le n` yau

Au cours du chapitre précédent, nous avons vu que les architectures des différents systèmes d'exploitation les plus répandus avaient des similitudes importantes. La ressemblance se poursuit concernant la gestion des extensions du noyau et, entre autres, les pilotes de périphériques. Une extension réalise une tâche pour une application que la distribution d'origine du noyau n'est pas à même de réaliser. Les systèmes d'interaction entre l'application et le pilote diffèrent, mais ils conservent deux éléments fondamentaux :

- Les applications interagissent avec le noyau via les fonctions de l'API, l'interface de programmation mise à disposition pour les applications.

- Les pilotes interagissent avec le noyau via les fonctions de la DPI (*Driver Programming Interface*), l'interface de programmation mise à disposition pour les pilotes.

Nous allons présenter, dans ce paragraphe, les composants de ce second point, l'interface entre les pilotes et le noyau. Ce sont les mécanismes de détection de cette interface qui permettent d'éviter une grande partie des propagations d'erreurs des pilotes vers le noyau, et au-delà vers les applications. L'approche de caractérisation que nous proposons consiste à évaluer le comportement du système en présence d'appels corrompus au niveau de l'interface entre les pilotes et le noyau.

3.1.1 Introduction à la DPI

Un module de gestion des périphériques fonctionne certainement dans le plus chaotique des environnements qui puisse exister dans un système d'exploitation, et la DPI en est le cœur. En effet, lors de la programmation d'un pilote, une simple opération d'entrée-sortie demande une séquence d'instructions particulières, propre au développement de code dans le cœur du système. Ces fonctions spécifiques ne sont pas impliquées dans la programmation des applications, car elles ne sont généralement pas nécessaires.

Plusieurs programmes applicatifs peuvent faire appel à un même pilote pour répondre à des opérations d'entrée-sortie, des interruptions matérielles, ou des événements temporels. Sur un système multiprocesseur, les processus associés peuvent fonctionner de manière concurrente. Une bonne maîtrise de l'ensemble de telles requêtes entrantes, chacune correspondant à un événement particulier, est particulièrement difficile à atteindre dans la pratique. Un pilote doit garantir l'accès exclusif pour n'importe quelle opération, de manière à protéger ses structures de données d'un éventuel accès concurrent. En effet, une telle situation pourrait amener à la corruption de données, voire un blocage du pilote.

Un autre des principaux rôles d'un pilote de périphériques est de répondre aux demandes issues des composants matériels. Ces dernières se manifestent généralement sous la forme d'une interruption. La gestion de ces interruptions est réalisée dans le noyau.

Pour réaliser ces différentes opérations, un pilote utilise des fonctions du noyau regroupées dans la DPI. Le noyau offre une liste de fonctions dédiées à la programmation du noyau et des extensions. Par exemple, dans le cas de *Windows XP*, la garantie d'un accès exclusif à des données est assurée par les méthodes de la classe *IOWorkLoop* et les classes d'événements associées. Sous *Linux*, l'implémentation est différente. Elle est constituée d'un ensemble de symboles (fonctions, structures, variables) disponibles uniquement pour le système d'exploitation : le noyau et les pilotes. Chacun des pilotes de périphériques peut augmenter cet ensemble de symboles.

Le Tableau 3-1 présente un ensemble restreint de services disponibles dans la plupart des DPI et des exemples de fonctions qui y sont associées. Toutes les DPI disposent de fonctions permettant l'allocation et la libération de la mémoire. Elles se nomment *kmalloc* et *kfree* sous *Linux*, *ExAllocatePool* et *ExFreePool* sous *Windows XP*, et enfin *MPAllocateAligned* et *MPFree* sous *Mac OS X*. Il en est de même pour les services d'enregistrement du pilote ou d'échanges de données. Toutefois, certaines fonctions ne sont pas présentes dans toutes les DPI. Sous *Linux*, les fonctions pour gérer les interruptions matérielles se nomment *request_irq,*

free_irq, irq_stat. Ces fonctions n'ont pas d'équivalents dans MacOsX, où la gestion des interruptions est masquée par l'environnement de développement des pilotes.

Service	**Exemples de fonctions associées**		
	GNU/*Linux*	***Windows XP***	**MacOS X**
Mémoire	*kmalloc, kfree*	*ExAllocatePool, ExFreePool*	*MPAllocateAligned, MPFree*
Interruption	*request_irq*	*oConnectInterrupt,, DforIsr*	*N/A*
Enregistrement	*register_X*	*IoregisterX*	*deviceMap*
Echanges	*Copy_{from, to}_user*	*RtlMoveMemory*	*MPBlockCopy*
« Bufferisation »	*N/A*	*SystemBuffer*	*N/A*

Tableau 3-1 Identification des fonctions des principaux systèmes d'exploitation

Chaque module du noyau ajoute un certain nombre de fonctions. Ces fonctions sont réservées aux processus privilégiés du système et composent ainsi la DPI. Nous allons voir dans les prochains paragraphes l'ensemble des services disponibles et la technique d'appel aux fonctions de l'interface pilote-noyau.

3.1.2 Interface générique aux systèmes d'exploitation

Le code des pilotes de périphériques, comme celui des applications, n'est pas autonome. Les deux codes utilisent diverses fonctions provenant du noyau du système. Mais, à la différence de l'API entre le système et les applications, la DPI est une interface dont la composition est parfois opaque. Nous avons ainsi cherché à définir cette interface de manière claire et le plus précisément possible pour que la description de cette interface entre le pilote de périphériques et le noyau soit portable d'un système à l'autre. Les fonctions identifiées dans la DPI sont classifiées par services dans le Tableau 3-2. Chaque service permet la réalisation d'une tâche par le noyau. Par exemple, le service de configuration du noyau permet d'étendre le noyau avec le code du pilote. L'insertion et la suppression du code du pilote dans le noyau sont réalisées par les fonctions appartenant à ce service. La liste n'est pas exhaustive, cependant, les principaux services utilisés par les extensions des noyaux les plus répandus sont représentés dans ce tableau. En effet, certains systèmes mettent en place des groupes de fonctions particulières permettant la réalisation d'actions spécifiques au système d'exploitation (voir chapitre 1).

Services	Rôle principal
Interface	Normalise et définit les points d'entrée
Configuration	Permet l'insertion et la suppression du pilote
Resources	Attributions des ressources pour le pilote
Error Handling	Gestion des erreurs
Buffer	Gestion des échanges de données
IMC	Communication entre les modules du noyau
I/O System	Gestion des ports d'E/S, mappage/libération mémoire, bus, DMA
Control Block	Gestion des blocs de données externes et internes
Memory	Gestion de la mémoire
Interrupt	Attribution et libération d'un canal d'interruption
Time	Mesures et observations liées au temps

Tableau 3-2 Classification des services de la DPI

La Figure 3-1 présente l'architecture logicielle des pilotes de périphériques des systèmes d'exploitation modernes. Les requêtes d'E/S effectuées par les applications sur le matériel sont relayées par le noyau vers le pilote. Ce dernier agit sur le périphérique directement ou via des fonctions de la DPI du plus bas niveau, permettant de gérer, par exemple, les bus ou la mémoire.

La DPI n'est pas le seul moyen de communication entre le pilote et les autres logiciels. Par exemple, il est possible, sur certains anciens systèmes, de lier directement le code d'un pilote sur une autre partie de code du noyau, grâce à des instructions particulières non accessibles aux applications. L'espace d'adressage du noyau abrite tous les processus privilégiés. Il est aisé d'accéder à des zones réservées du noyau. Toutefois, ces canaux d'accès ont été renforcés ces dernières années sur plusieurs systèmes, comme *Windows XP* et *Mac OS X,* grâce à un environnement de développement adapté et au respect de contraintes de programmation. L'objectif est d'obtenir une amélioration de la qualité du code des logiciels correspondant aux extensions du noyaux, notamment les pilotes.

La DPI peut être plus ou moins accessible en fonction du système d'exploitation, mais, dans les systèmes les plus répandus, on retrouve les mêmes services fournis par le noyau (cf. Tableau 3-2). Ils se composent essentiellement de la gestion des temporisateurs (*timers*), des interruptions, de la mémoire et de l'enregistrement du pilote dans le système. Les pilotes implémentant des services particuliers, comme le *Plug-And-Play* ou l'alimentation, doivent également utiliser certaines ressources du noyau pour mettre en œuvre les routines adéquates.

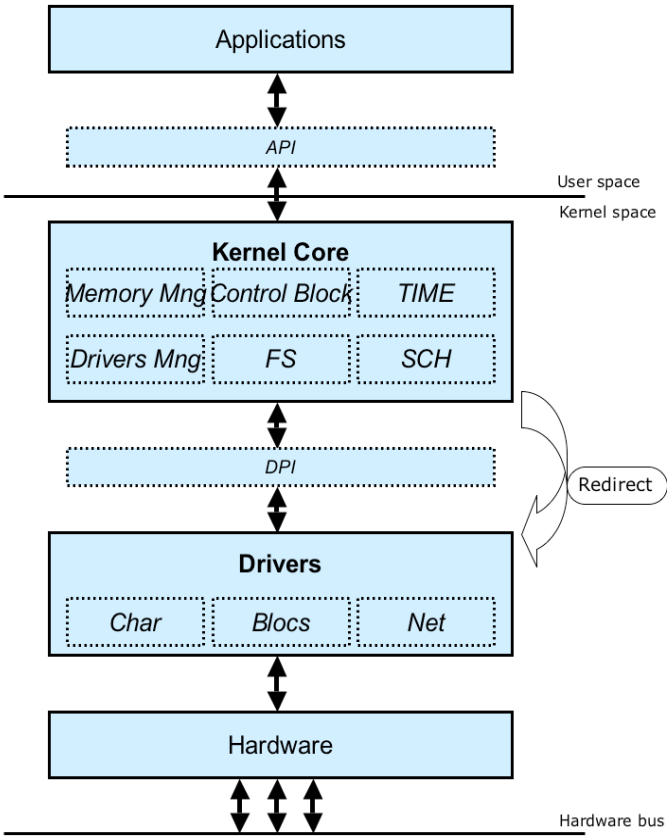


Figure 3-1 Architecture d'un système d'exploitation moderne

3.1.3 Le paramétrage des fonctions privilégiées

Dans tous les systèmes, les modules de gestion de périphériques font appel à des fonctions réservées, mises à disposition par le noyau. Nous allons voir au cours de ce paragraphe comment s'effectue le paramétrage de ces fonctions.

Comme les fonctions ordinaires, les fonctions de l'interface pilote noyau nécessitent souvent des paramètres d'entrée et/ou de sortie. Ils sont généralement composés des valeurs courantes (i.e., numériques), des constantes ou des valeurs d'adresse. En fonction du système et de la fonction, le nombre de paramètres varie de manière importante : certaines fonctions n'utilisent pas de paramètres d'entrée/sortie alors que d'autres mettent en œuvre de nombreux paramètres.

L'interface pilote noyau constitue, par son statut de canal de communication, le canal privilégié de propagation des erreurs des pilotes vers le noyau. Cependant, dans la plupart des architectures actuelles, les pilotes peuvent corrompre le noyau directement en agissant sur son espace d'adressage, sans faire usage de la DPI, en liant les programmes à l'intérieur du noyau. Les analyses en opération montrent que les conséquences des fautes logicielles sont souvent révélées au niveau des canaux de communication.

Les structures et variables du noyau sont souvent modifiées par les fonctions de la DPI. Une erreur de paramétrage lors d'un appel à une fonction réservée du noyau peut avoir des

conséquences catastrophiques sur l'état du système. Les résultats qu'il est possible d'obtenir à partir de nos travaux permettent de mieux appréhender cet impact.

3.2 Critères p`ur l'évaluati`n de l'impact des pil`tes défaillants

Nous avons vu dans le paragraphe précédent que l'interface pilote noyau constitue un élément particulièrement intéressant pour l'étude des systèmes vis-à-vis des pilotes composants leur environnement. Durant les deux premiers chapitres, nous avons évoqué la nécessité pour un étalon de sûreté de fonctionnement d'être reconnu par la communauté. Les critères pour obtenir cette reconnaissance par rapport à un travail sur la DPI sont rassemblés dans ce chapitre. En effet nous rappelons dans ce paragraphe les origines des fautes que nous avons souhaité simuler et étudier. Nous présentons les contraintes de conception d'un étalon permettant une portabilité importante de la méthode à un large spectre de systèmes existants. Nous présentons quelles mesures nous souhaitons mettre en évidence par nos travaux puis nous situons nos recherches dans la définition d'un cadre pour la conception des étalons de sûreté de fonctionnement des systèmes informatiques notamment initié dans le projet Européen DBench [Kanoun *et al*. 2002].

3.2.1 Fautes visées

Dans le premier chapitre, nous avons présenté les difficultés de la programmation des pilotes de périphériques. Durant nos travaux, nous nous sommes concentrés sur les fautes les plus fréquemment à l'origine d'une défaillance d'un pilote de périphériques : les fautes résiduelles du logiciel.

En effet, comme nous l'avons déjà souligné, la conception d'un pilote de périphériques n'est pas triviale : elle doit allier performance, sûreté de fonctionnement et compatibilité avec les différentes versions de matériels et de systèmes. Aussi, les développeurs de logiciels de pilotages doivent combiner des connaissances dans des domaines très distincts. De plus, les programmeurs ne sont pas les seuls en cause, les spécifications pour la conception d'un pilote sont souvent informelles et confuses. La terminologie propre au domaine, les limites du langage humain et le mélange des niveaux d'abstraction constituent des difficultés supplémentaires lors du développement d'un logiciel de pilotage. L'utilisation de moyens, tant traditionnels que non conventionnels, de conception de logiciels montre ces limites. Les développeurs de pilotes des fabricants de matériel (comme des contrôleurs ou des périphériques) n'ont pas la même expérience du développement interne au noyau que les fournisseurs de logiciels d'exploitation. Pourtant, la compréhension globale de la gestion des périphériques par un noyau peut s'avérer cruciale lors de la programmation de pilotes, et les documentations sont parfois confuses et destinées à un public très averti. De plus, même si ces difficultés sont surmontées, le code des pilotes n'est pas exsangue de fautes.

En effet, les fautes résiduelles lors de l'élaboration d'un logiciel sont inéluctables. Aucun programme complexe ne peut garantir d'être exempt d'erreurs. Les extensions font partie des logiciels fortement touchés par ce phénomène, comme présenté dans le deuxième chapitre. Par exemple, des pilotes sont fréquemment élaborés en s'appuyant sur des "canevas" mal adaptés, voire en copiant et en éditant le code de pilotes existants, souvent sans compréhension

complète. Ce genre de pratique conduit à des bogues subtiles, difficilement identifiables, car la présence des pilotes au cœur du noyau peut conduire à des problèmes souvent difficilement analysables par le développeur de pilote.

Les techniques actuelles de conception de pilotes de périphériques comportent de grosses lacunes en terme de qualité de logiciel. Les conditions de développement sont souvent très difficiles et il en résulte que les fautes logicielles résiduelles sont la principale cause d'erreurs liées aux extensions. Toutefois, les fautes matérielles représentent une deuxième source d'erreurs. Aujourd'hui faible, leur part risquerait d'augmenter avec la venue d'amélioration de la prévention de fautes logicielles. Ces types de fautes, externes au noyau, sont aujourd'hui responsables d'un grand nombre de défaillances des systèmes d'exploitation. Nous avons souhaité évaluer le comportement du système en terme de sûreté de fonctionnement vis-à-vis de ces fautes.

3.2.2 Portabilité de la méthode

Les systèmes principalement ciblés par nos travaux sont les systèmes d'exploitation à usage général comme *Windows* ou GNU/*Linux*. Le principal objectif de la caractérisation est la comparaison des résultats obtenus lors de l'évaluation du comportement de plusieurs systèmes d'exploitation distincts, dans notre cas vis-à-vis de pilotes défaillants. Nous allons discuter dans ce paragraphe des contraintes que nous nous sommes fixées pour établir une approche standard des systèmes d'exploitation modernes. En effet, durant le chapitre 2, nous avons vu que la reconnaissance du domaine est essentielle pour la crédibilité de l'approche et l'obtention de critères reconnus avec des mesures comparables.

La possibilité de disposer d'une description détaillée du système cible permettrait d'effectuer des analyses plus précises mais en contrepartie elle en réduirait le portage. Il convient ainsi de conserver une portabilité importante de la méthode d'évaluation. Comme présenté dans le premier chapitre dans les dimensions de catégorisation, nous avons donc opté pour une approche globale du système en prenant le noyau comme cœur du système d'exploitation : il assure le lien entre les applications et le matériel en se basant, entre autres, sur les logiciels de pilotage des périphériques. C'est la sûreté de fonctionnement de ce noyau minimal que nous cherchons à évaluer au cours de nos travaux. Toutefois, le noyau pourra être l'objet d'une analyse plus précise, par exemple en analysant chacune de ses parties, pour obtenir des résultats plus précis sur l'évolution d'un noyau.

D'autre part, nous considérons que l'utilisateur de l'étalon est différent du développeur du système d'exploitation cible. C'est pourquoi nous supposons :

- une connaissance limitée de la structure interne du système ;
- un accès limité aux codes des pilotes et du noyau.

Cette vision du système est proche de celle appelée couramment « boîte noire ». Nous considérons que nous ignorons les mécanismes internes au composant ciblé, et qu'en particulier nous ne disposons pas de son code source. Ce point de vue assure une meilleure portabilité de la méthode à différents systèmes. En effet, l'architecture interne du noyau n'étant pas prise en compte, les techniques d'évaluation peuvent être plus aisément réutilisées, sur des systèmes différents. Au niveau de la connaissance des codes source des systèmes cibles, cette

approche permet de s'intéresser de manière indifférente à des systèmes à code source ouvert ou non.

Nous avons étendu cette vision opaque du code source au niveau des pilotes. En effet, la prise en compte du contenu des pilotes de périphériques dans une méthode d'étalonnage des systèmes pourrait faire émerger des difficultés lors du portage à des systèmes d'exploitation distincts. En effet, des systèmes d'exploitation comme *Linux* et *Windows* utilisent des pilotes de périphériques aux codes sources radicalement différents. L'approche « boîte noire » d'un composant permet de s'affranchir de prise en compte de la structure interne de celui-ci. Ses interfaces avec son environnement restent les seuls éléments connus. C'est le cas des interfaces d'un noyau de système d'exploitation, qui sont connues du public et constituent donc une localisation particulièrement intéressante pour l'étude de l'impact des pilotes de périphériques sur le reste du système.

3.2.3 Des mesures appropriées

Nous avons observé dans le paragraphe précédent que les interfaces représentaient des localisations privilégiées pour la caractérisation d'un système d'exploitation. En effet, elles font partie du système et composent la partie visible pour son environnement (les applications et les pilotes). Comme nous l'avons vu au chapitre 2, la caractérisation d'un composant vis-à-vis de fautes externes permet l'évaluation des mesures de robustesse. Nous verrons, dans ce paragraphe, la possibilité d'élargir les mesures obtenues au-delà des mesures de robustesse.

Dans notre approche, l'environnement opérationnel potentiellement défaillant concerne les logiciels de pilotage des périphériques du système d'exploitation, les effets des applications et du matériel étant évalués dans d'autres travaux. Cependant, une même erreur peut avoir des origines distinctes, les fautes dans les pilotes peuvent alors être représentatives de fautes provenant du matériel. Nous reviendrons sur la représentativité des fautes à la fin de ce chapitre.

Nous avons vu que la caractérisation du système à fonctionner correctement en présence d'entrées invalides ou de conditions environnementales stressantes forme une mesure de robustesse particulièrement intéressante. Plus concrètement, elle s'établit par l'observation de deux classes d'événements :

- Les modes de défaillance du système lorsque l'erreur est propagée au système cible. Ils correspondent ici à l'état des applications s'exécutant sur le système.
- Les mécanismes de détection ou de tolérance quand l'erreur est confinée au niveau de l'interface ou à l'intérieur du système. Ils sont généralement composés des mécanismes de notification d'événements internes et des codes retour des fonctions composant l'interface visée par les entrées erronées.

Cependant, une évaluation au niveau d'une interface permet également la mesure efficace d'une donnée essentielle aujourd'hui : le temps. Les mesures fournies par les étalons de performance permettent la comparaison de la performance de certains mécanismes et services fournis par les systèmes d'exploitation. Aucun de ces étalons ne fournit de mesures de sûreté de fonctionnement. Cependant, il est possible de les réutiliser dans un contexte d'étalonnage de

sûreté de fonctionnement pour pouvoir mesurer la « dégradation » de la performance en présence de fautes.

En effet, le temps de réponse à une requête formulée via les fonctions du système est une information très importante pour l'évaluation du service d'un système d'exploitation vis-à-vis de défaillances applicatives (voir [Kalakech *et al.* 2004] et [Koopman *et al. 1997*]). De manière similaire, l'analyse de la dégradation temporelle des services du système en présence d'extensions défaillantes est une donnée intéressante pour la comparaison des systèmes. La meilleure méthode pour évaluer l'impact des pilotes de périphériques sur les services du noyau est l'intégration d'une charge applicative témoin s'exécutant pendant l'injection de fautes. Nous verrons la composition de cette charge dans la partie implémentation décrite dans le prochain chapitre.

3.2.4 Compléter les étalons existants : DBench

Nous avons vu durant le deuxième chapitre que la caractérisation des systèmes d'exploitation peut être effectuée en prenant en compte trois origines distinctes de perturbation : les fautes logicielles provenant des applications, les fautes provenant matériel et, dans le cas nous concernant, les fautes logicielles résiduelles dans les pilotes de périphériques.

Un concepteur de système à base de COTS souhaite obtenir des informations aussi objectives que possible sur la sûreté de fonctionnement du composant qu'il souhaite intégrer, c'est-à-dire le noyau, et non celle qui concerne un ensemble incluant ce composant, comme le noyau étendu. En effet, le système à base de COTS qui intégrera le composant évalué et le composant lui-même ne disposent probablement pas des mêmes architectures et périphériques matériels sous jacents. Les mesures de sûreté de fonctionnement doivent donc considérer principalement le noyau minimaliste du système d'exploitation et cibler le comportement de ce COTS vis-à-vis de son environnement potentiel. Les travaux effectués dans [Kalakech *et al.* 2004] permettent d'évaluer l'impact sur le comportement du système des fautes provenant des applications. Nous complétons ces travaux en explorant l'environnement potentiel des pilotes et en élaborant une méthodologie permettant de distinguer le noyau de ses extensions. Ces résultats intéressent les développeurs de noyaux car, si la distinction des erreurs provenant des pilotes et du noyau est établie, la responsabilité du noyau pourrait être écartée dans de nombreux cas, libérant les gros groupes industriels comme *Microsoft* et *Apple* d'une part de la responsabilité des défaillances des systèmes d'exploitation en opération.

Ce phénomène est accentué pour les raisons déjà évoquées au début du premier chapitre. Les pilotes sont une source d'erreurs très importante et leur prise en compte est aujourd'hui essentielle. À l'inverse, les noyaux des systèmes d'exploitation bénéficient d'une expérience de retour importante. Le noyau *Linux* a disposé de plus de 15 ans de fonctionnement en opération pour améliorer et corriger les nombreux bogues résiduels que comportait sa version initiale. Il en est de même pour Microsoft, la firme américaine distribue son logiciel *Windows* depuis maintenant une quinzaine d'années. Les programmeurs de noyaux de systèmes d'exploitation sont donc des personnes souvent très qualifiées et disposant d'une grande expérience. Les fautes provenant du noyau et de ses extensions dans le cadre d'un étalon de sûreté de fonctionnement d'un système d'exploitation doivent donc être distinguées. C'est dans cet objectif que nos travaux s'intègrent dans le cadre du projet DBench.

3.3 Évaluati` n de la r` bustesse de la DPI

L'évaluation de la robustesse de l'interface DPI du noyau représente un point de vue nouveau dans le cadre de l'étalonnage de la sûreté de fonctionnement des systèmes d'exploitation. Dans ce paragraphe, nous présenterons une méthode permettant de :

- Simuler des pilotes défaillants, fautes considérées externes au noyau dans notre étude.
- Conserver une portabilité importante de la méthodologie.
- Obtenir des mesures reconnues et adaptées aux besoins de sûreté de fonctionnement des systèmes à base de COTS.
- Compléter les travaux de conception d'un cadre pour l'étalonnage de la sûreté de fonctionnement des systèmes d'exploitation.

Dans ce paragraphe, nous expliquons notre choix d'une technique d'injection de fautes par logiciel adaptée à nos besoins, en discutant de la représentativité de cette approche. Nous décrivons ensuite le rôle de la charge de travail. Nous présentons enfin les différents niveaux d'observations et les mesures qu'ils permettent d'extraire.

La Figure 3-2 résume cette approche. Le noyau représente le composant ciblé par l'étalon. La charge de travail se situe au niveau des applications. Elle a pour rôle d'effectuer des opérations, via l'API, sur les périphériques du système. Le noyau relaye la requête vers le pilote et celui-ci répond à la demande des applications. Pour cela, il exécute des fonctions de la DPI. La corruption des paramètres est effectuée à cet instant-là, sur un paramètre d'une fonction appelée par le pilote « fautif ». Les fautes sont regroupées dans la charge de fautes. Les observations se situent à deux niveaux : Au niveau de la DPI par l'observation du code retour de la requête du pilote, et au niveau de l'API par l'observation du comportement de la charge et du système. L'ensemble formé par le noyau, ses extensions et le matériel constitue ce que l'on nomme la machine ciblée, c'est-à-dire le système concrètement évalué par l'étalon. En effet, il est impossible d'évaluer expérimentalement un système sans y associer le support d'exécution procuré par l'architecture matérielle sous-jacente.

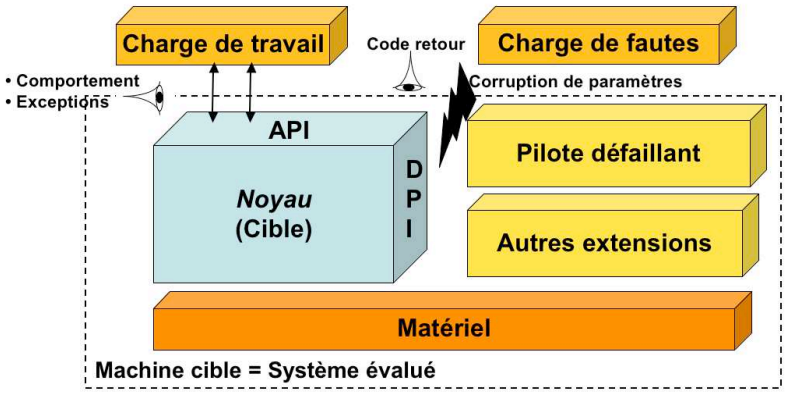


Figure 3-2 Test de robustesse de la DPI d'un noyau

3.3.1 Technique d'injection de fautes par logiciel

Dans cette section, nous justifions le choix de la technique d'injection de fautes par logiciel utilisée puis nous discutons de la représentativité des fautes injectées, un des critères essentiels à l'acceptation d'une technique d'injection de fautes par logiciel.

3.3.1.1 Tests de robustesse

La robustesse d'un composant est mesurée grâce à l'observation de son comportement en présence de fautes externes lors d'un processus d'étalonnage. Notre approche présente un test de robustesse de l'interface entre les extensions et le noyau. Ce test permet de caractériser le comportement du système d'exploitation vis-à-vis de pilotes défaillants.

Un des critères de notre approche présenté au début de ce chapitre impose un aspect « boîte noire » du système et du pilote. Donc l'injection de fautes ne peut être effectuée sur le code du système ou du pilote avant sa mise en opération. Elle doit être réalisée lors de l'exécution du processus noyau, durant le flot d'exécution des instructions liées au code du pilote. Pendant l'exécution, un mécanisme va donc déclencher l'injection de fautes. Pour cela, le déclencheur utilise un temporisateur ou une condition exceptionnelle.

Une expérience d'injection de fautes consiste à appliquer une faute à un système cible soumis à une activité donnée. Des relevés, effectués au cours de l'expérience, permettent de déduire des mesures de sûreté de fonctionnement relatives au système et à ses mécanismes de tolérance aux fautes. L'utilisation de l'injection de fautes nécessite de répondre au préalable aux questions suivantes [Arlat *et al.* 1990] :

- Quelles sont les fautes à injecter, comment les injecter et quelle est l'activité du système à prendre en considération (ce qui revient à définir le domaine d'entrée)
- Quels comportements sont à observer et comment traiter ces observations (ce qui définit le domaine de sortie). Le domaine d'entrée est caractérisé par l'ensemble de fautes à injecter F et l'ensemble A qui spécifie l'activité du système cible. Le domaine de sortie correspond à l'ensemble d'observations R et l'ensemble de mesures M dérivé après un traitement des ensembles F, A et R.

Nous montrerons dans les prochains paragraphes comment nous avons choisi d'y répondre.

3.3.1.2 Corruption de paramètres

Nous venons de présenter les avantages de la technique d'injection de fautes par logiciel pour caractériser la robustesse d'un système d'exploitation vis-à-vis d'un pilote défaillant. Toutefois, différentes perturbations peuvent être injectées. Deux d'entre elles ont particulièrement attiré notre attention pour simuler une faute propagée au niveau d'une interface :

- L'inversion de bit (dans le code ou les paramètres),
- Corruption de paramètre par une valeur corrompue spécifique.

Dans le cas qui nous concerne, les fonctions appelées constituent en général des accès à des services du noyau. Ces fonctions sont généralement composées d'un grand nombre de paramètres (particulièrement sous *Windows*), dont, de plus, certains peuvent paraître obscurs pour une personne non-initiée en programmation de pilotes. Les deux perturbations sont donc particulièrement adaptées pour simuler une faute logicielle résiduelle dans les pilotes de périphérique.

Ces paramètres sont communs à la plupart des interfaces des systèmes d'exploitation. Pour représenter un mauvais paramétrage d'une fonction, une simple inversion de bit aléatoire suffit. Toutefois, l'analyse d'une telle injection est rendue assez complexe car l'effet réalisé sur la machine n'est pas toujours explicite. L'interprétation se limite souvent à l'observation des modes de défaillances du système [Jarboui *et al.* 2003].

À l'opposé, les techniques de corruption avec des valeurs erronées comprises dans l'espace de validité du paramètre sont plus souples et permettent une analyse plus rapide et plus efficace des comportements du système. L'avantage au niveau de la mise en place et du gain de temps est indéniable. Toutefois, elles nécessitent une importante étude au préalable sur les valeurs à corrompre. La Figure 3-3 montre comment simuler des fautes logicielles par une technique de corruption de paramètres. En se basant sur le travail réalisé dans le cadre de Ballista [Koopman & DeVale 1999] qui visait l'API (voir chapitre 1), nous avons regroupé les paramètres des fonctions du noyau dans plusieurs classes. Ces classes sont dépendantes du type de variables utilisées dans chaque système d'exploitation. Pour chaque classe de types de paramètres, nous avons conçu lors d'une étape préalable un ensemble de valeurs qui dépendent essentiellement de la sémantique du type. Ces valeurs sont généralement les bornes et les valeurs spécifiques du domaine de validité.

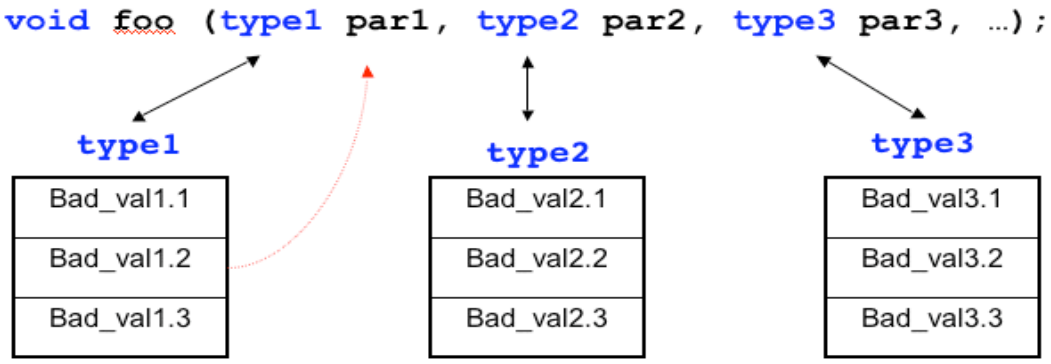


Figure 3-3 Principe de corruption des paramètres d'une fonction

Un aspect très important concerne les conséquences des fautes simulées : les erreurs. En pratique, une même erreur peut être provoquée par des fautes différentes. Cette constatation a été le point de départ des techniques d'injection de fautes par logiciel. Comme le présente la Figure 3-4, les interfaces (API, DPI et matérielle) forment des goulots pour la propagation des erreurs. Par exemple, deux erreurs dans un pilote, que leurs origines soient de natures physiques ou logicielles, peuvent engendrer une même erreur au niveau de la DPI. Ces deux erreurs affecteront de manière équivalente le noyau. Par ailleurs, l'erreur isolé au niveau du

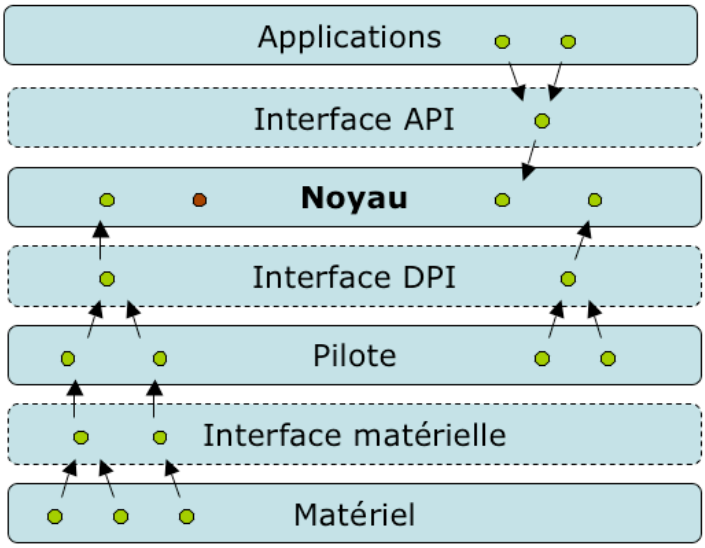


Figure 3-4 Propagation d'erreurs dans un système d'exploitation

noyau (notifiée de couleur différente) sur la figure représente la possibilité que certaines erreurs dues à une faute interne au noyau ne peuvent être simulées à partir de l'environnement du noyau. Enfin, la figure illustre également les résultats publiés dans [Jarboui *et al.* 2002a], que nous avons déjà abordés, montrant que les erreurs simulées au niveau de l'API ne pouvaient pas être à l'origine d'un nombre important d'erreurs présentes dans le noyau et relevées dans les journaux d'erreurs des systèmes d'exploitation en opération. Ces erreurs non simulables au niveau de l'API proviennent en grande partie des pilotes et également, pour une moindre part, du noyau du système.

3.3.1.3 Représentativité de l'injection de fautes

Nous avons vu que dans notre évaluation du système d'exploitation du point de vue des pilotes de périphériques, nous caractérisions le système vis-à-vis d'un ensemble de faute externes. Une des principales difficultés à l'obtention d'un standard reconnu pour un étalon reste la reconnaissance des cas de fautes étudiés. En effet, compte tenu de l'importance de cette dimension, une étude concernant la représentativité des fautes logicielles est nécessaire pour avoir confiance dans les résultats retournés par les expériences d'injection de fautes. Nous discuterons dans ce paragraphe de l'importance de la représentativité des fautes dans le contexte d'étalonnage de sûreté de fonctionnement.

Nous avons vu dans le chapitre deux que, en pratique, une même erreur peut être provoquée par des fautes différentes. Les techniques d'injection de fautes par logiciel se basent sur cette constatation pour justifier les fautes injectées. Plusieurs études montrent que l'inversion des bits dans les registres du processeur ou dans la mémoire permet de simuler des fautes matérielles transitoires, en partie certaines fautes logicielles [Costa *et al.* 2001]. De plus, les erreurs provoquées sont similaires à celles relevées lors de campagnes d'injections physiques de fautes (voir [Fuchs 1996] et [Christmansson *et al.* 1998]).

Actuellement, aucun modèle de fautes n'a été adopté unanimement pour représenter les fautes logicielles. Le modèle ODC[16] [Chillarege *et al*. 1992] rapporte un certain classement de types de fautes qui ont été observés lors de la phase de conception et en exploitation de grands systèmes d'exploitation développés par IBM. La technique décrite traite les informations contenues dans les relevés en opération et durant le processus de développement du logiciel lors de la manifestation des erreurs de programmation. Ces données sont alors employées pour la réduction des coûts, l'amélioration de la qualité du logiciel, l'ordonnancement des tâches, le diagnostic...

Plusieurs études sur la représentativité des fautes injectées ciblent les fautes d'origine physique. Ces travaux mettent en évidence la représentativité des techniques d'injection de fautes par logiciel vis-à-vis des fautes physiques [Czeck 1993] [Kanawati *et al*. 1995] [Rimén *et al*. 1994] [Fuchs 1998] [Cheynet *et al*. 2000]. Par ailleurs, mis à part quelques études ponctuelles [Christmansson & Chillarege 1996, Madeira *et al*. 2000], moins d'attention a été portée à la représentativité des fautes logicielles, qui constitue pourtant une des principales causes de défaillances des systèmes actuels [Lee & Iyer 1995].

Le travail présenté dans [Jarboui *et al*. 2003] a pour but de définir un ensemble d'expériences globales et coordonnées. Les résultats de ces expériences fournissent une bonne évaluation des fautes effectivement couvertes par les techniques actuelles d'injection de fautes. Un des résultats intéressant pour nos travaux concerne le modèle de fautes le plus approprié à associer aux mesures de robustesse. Les résultats présentés dans ces travaux comparent trois techniques d'injection de fautes :

- Paramètres invalides simulant une faute externe
- Inversion de la valeur d'un bit simulant une faute externe
- Inversion de la valeur d'un bit simulant une faute interne

L'étude montre que les modes de défaillances mis en évidence par les deux premières techniques sont identiques entre elles mais différentes de la troisième. Toutefois, le nombre d'expériences à mettre en œuvre pour obtenir ces résultats est très différent entre la technique de corruption de paramètre par valeurs erronées et la technique d'inversion de bits simulant une faute externe. L'étude préconise donc l'utilisation de la technique des paramètres invalides pour l'obtention de mesures de robustesse par rapport à un logiciel erroné. De plus, les fautes injectées sont proches des conséquences des fautes logicielles réelles des applications.

La définition de l'ensemble de fautes est inévitablement un processus pragmatique très complexe. Le retour d'expérience dans plusieurs domaines d'application est nécessaire pour accomplir cette tâche avec succès. Par exemple, les informations concernant les fautes révélées en vie opérationnelle, et les études expérimentales ou de simulation permettent d'établir des modèles de fautes de référence.

---

[16] *Orthogonal Defect Classification*

3.3.1.4 Charge de faute

Lors de l'évaluation de la robustesse d'un système, chaque paramètre est testé avec plusieurs valeurs possibles. Des exemples des valeurs injectées sont présentés dans le Tableau 3-3.

Type du paramètre corrompu	Valeurs injectées
Entier	*INT_MIN*
	0
	INT_MAX
Longueur	*L_MIN*
	0
	L_MAX
Descripteur de fichiers	*D_MIN*
	0
	D_MAX
Identificateur de processus	*Id_MIN*
	0
	1
	Id_MAX
Mode	*WO (Write Only)*
	RO (Read Only)
	RW (Read Write)
	RWE (Read Write Execute)
	E (Execute)

Tableau 3-3 Exemple de valeurs injectées lors de la corruption d'un paramètre

Les paramètres dépendent du noyau ciblé. Les valeurs peuvent changer selon les plates-formes logicielles et matérielles, car les paramètres peuvent être définis sur des intervalles différents. Le choix des valeurs de corruption pour effectuer un test exhaustif de robustesse est alors affecté. Toutefois, les valeurs corrompues sont toujours choisies parmi les valeurs les plus pertinentes de l'intervalle de définition, c'est-à-dire :

- Les bornes de l'intervalle de définition, comme les valeurs *INT_MIN* et *INT_MAX* pour l'intervalle de définition des entiers,
- Les valeurs spécifiques, par exemple la valeur *0* pour un entier ou la valeur *NULL* pour un pointeur.

3.3.2 Charge de travail

Les étalons de performance constituent une des premières sources pour le choix de l'activité à appliquer au système pendant son étalonnage. C'est notamment le cas lorsque les mesures recherchées sont des mesures temporelles. Nous avons vu dans les concepts abordés dans le deuxième chapitre que l'activité représente une partie essentielle des profils d'exécution pour

la validité des mesures d'étalonnage de la sûreté de fonctionnement des systèmes d'exploitation.

Si le domaine d'application est connu et l'application existe, l'activité peut inclure l'application elle-même : c'est le cas de figure idéal. Mais dans le cas où le domaine d'application n'est pas disponible, une alternative consiste à déduire l'activité à partir de la spécification de l'application réelle sous la forme d'une trace d'exécution. Une telle trace fournit une activité du système qui est représentative du domaine d'application. Cette trace a comme rôle d'activer le système d'une manière similaire à une application réelle, mais elle ne permettra pas de donner un aperçu sur les modes de défaillance des applications. C'est pourquoi, nous proposons l'utilisation d'une application dont on connaît les résultats fournis à l'avance.

Afin de provoquer l'activation de la DPI par les pilotes et de simuler un comportement opérationnel réaliste, nous devons utiliser un procédé indirect d'activation au moyen d'une charge de travail appliquée au niveau de l'API. La Figure 3-5 illustre cette idée.

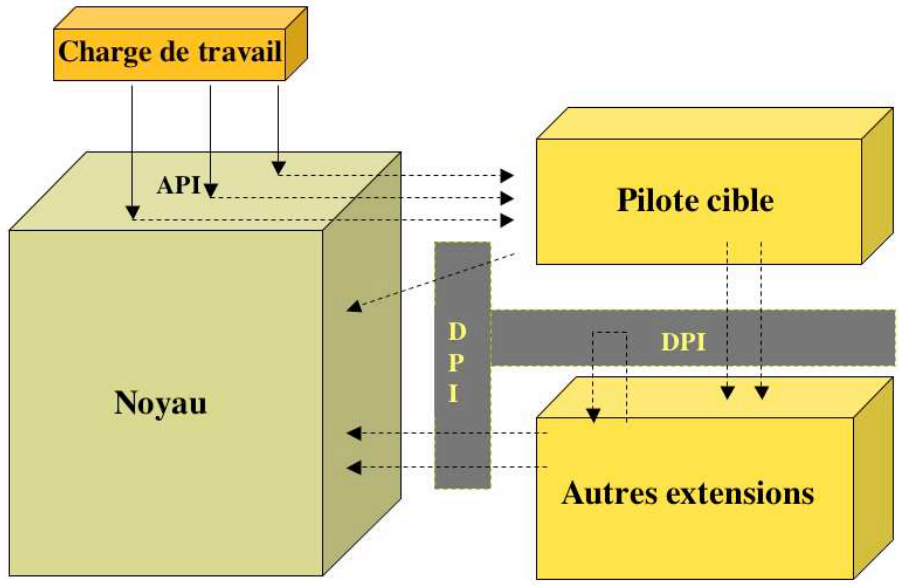


Figure 3-5 Profil d'exécution

La charge de travail doit effectuer des opérations activant les opérations implémentées par le pilote de périphériques. L'utilisation de l'ensemble des fonctionnalités d'un pilote permet d'activer la totalité des fonctions du noyau utilisées par ce pilote. En effet, chaque fonctionnalité du pilote utilise des fonctions du noyau pour réaliser la tâche qui lui est dévolue. La charge de travail activera donc indirectement l'utilisation des fonctions de l'interface entre les pilotes et le noyau.

Pour la réalisation des certaines opérations, un pilote peut faire appel à des fonctions extérieures implémentées par d'autres extensions ou pilotes. Ne faisant pas, par définition, partie de l'interface propre au noyau, ces fonctions ne seront pas étudiées lors de l'évaluation de la robustesse d'un noyau et sont exclues de la notion de DPI, propre au noyau.

Plus précisément, chaque processus au niveau applicatif effectue un ensemble d'opérations élémentaires sur un composant matériel spécifique dans l'objectif de (dans l'ordre) :

- Désinstaller le logiciel de pilotage du composant ciblé (s'il est déjà employé par le système),
- (Ré-)Installer le logiciel de pilotage du composant ciblé,
- Effectuer une série de demandes pour activer les autres fonctionnalités du pilote de périphériques. Par exemple, dans le cas d'une carte réseau, le processus applicatif teste le réseau Ethernet.
- Désinstaller le logiciel de pilotage du composant ciblé pour vérifier que le composant peut être correctement retiré du système.
- Réinstaller le pilote en particulier s'il est nécessaire au système (par exemple, une carte graphique ou une carte réseau).

Afin d'évaluer au mieux l'impact d'une défaillance sur l'ensemble du système, la charge de travail peut être exécutée une deuxième fois mais sans injection de fautes. Cela permet également d'évaluer les capacités du système à recouvrir une faute. C'est particulièrement utile pour améliorer le diagnostic dans le cas où aucun événement ne serait relevé lors de la première exécution en présence de faute.

3.3.3 Observations

L'objectif de l'approche que nous proposons est de fournir un ensemble d'observations adaptées à la mesure de la sûreté de fonctionnement d'un système d'exploitation en présence de pilotes défaillants. En conséquence, pour élaborer des analyses pertinentes, il est nécessaire d'obtenir une bonne variété d'observations.

Nous avons donc considéré deux niveaux d'observation :

- Le **niveau interne** concerne les perturbations perçues dans le système d'exploitation. Le principal élément de cette observation sera le possible code d'erreur retourné par la fonction corrompue par l'injection. Le relevé d'observations internes au noyau sont difficiles à concilier dans le respect des concepts fixés dans le paragraphe 3.2 : portabilité et faible intrusion. En effet, ces observations pourraient influer sur le comportement du système en alourdissant la charge de travail du noyau et en la modifiant. C'est dans cet objectif que nous nous sommes limités à la simple observation, lorsqu'elle est possible, du code retour d'erreur de la fonction.
- Le **niveau externe** a pour objectif de caractériser le comportement du système perçu par l'environnement externe au système d'exploitation : l'utilisateur d'une part, en comparant, par exemple, les résultats obtenus en présences de fautes avec les résultats obtenus en l'absence de fautes, et les applications d'autre part avec une observation au niveau de l'API du noyau. Il inclut la relève des erreurs rapportées par le noyau aux processus applicatifs de la charge de travail (exceptions, codes d'erreur des appels systèmes...).

Les observations spécifiques faites à chaque niveau sont décrites dans le Tableau 3-4. Ces niveaux d'observations peuvent être enrichis par une perception mieux adaptée à l'utilisateur au moyen d'observations directement liées aux applications (par exemple, le temps d'exécution de la charge de travail ou le temps de redémarrage du système).

Niveau	Événements observés
Interne	**Code retour d'erreur (EC)**: Code d'erreur retourné au pilote par le noyau au niveau de la DPI
Externe	**Exception (XC)**: Exceptions du processeur transmises par le système et observées au niveau de l'API **Blocage du noyau (KH)**: Le système d'exploitation ne répond plus aux requêtes du matériel et de l'API. La remise en opération du système nécessite une intervention spécifique. **Interruption de la charge (WA)**: La charge de travail a été interrompue : certains services au niveau de l'API n'ont pu être effectués. **Terminaison incorrecte de la charge (WI)**: La charge de travail s'est terminée, tous les services ont pu être effectués au niveau de l'API, mais certains retournent un code retour d'erreur.

Tableau 3-4 Niveaux d'observations du système et événements observés

Le code d'erreur retourné par une fonction du noyau constitue fréquemment la première observation possible de l'impact de la faute sur le comportement interne du noyau. Du point de vue de la robustesse, le retour d'un code d'erreur est essentiel. Les fonctions du noyau devraient réagir à un appel contenant un argument avec une valeur corrompue. La meilleure réaction du point de vue de la diagnosticabilité étant un code retour d'erreur correspondant à la faute injectée.

Si une exception matérielle du processeur (*div0* ou *Segfault* par exemple) est relevée lors de l'exécution d'un programme dans l'espace d'adressage du noyau, le noyau l'interrompt ou entre dans un mode dégradé.

La considération des événements liés à la charge de travail (les évènements WA ou WI dans le tableau) permet de mettre en évidence les canaux de propagation d'erreurs. On peut distinguer trois états de la charge de travail :

- La charge n'est pas affectée et fournit des résultats corrects. L'erreur provoquée par la faute injectée n'a pas été propagée au niveau applicatif.
- La charge est affectée et fournit des résultats incorrects. L'erreur provoquée a été propagée au niveau applicatif.
- La charge est interrompue et n'a pas fourni de résultats. L'erreur a été propagée au niveau applicatif ou à un des services du noyau essentiel au support de l'application.

La seconde exécution de la charge de travail permet, pour chaque expérience, d'évaluer les capacités du système à retourner dans un état stable. On peut distinguer trois possibilités :

- La seconde exécution de la charge de travail se déroule normalement. Le système est dans un état stable sans le recours à une intervention spécifique. C'est le cas le plus positif pour la charge.
- La seconde exécution de la charge de travail ne peut s'exécuter normalement et le système doit être redémarré pour revenir à un état stable.
- La seconde exécution de la charge de travail n'a pu être lancée. Le système n'est pas opérationnel. Une opération de maintenance est nécessaire pour redémarrer le système.

Un blocage noyau est diagnostiqué quand les demandes effectuées via les interfaces du noyau ne reçoivent pas de réponses dans un délai raisonnable, fixé par retour d'expérience. Ce délai définit la fenêtre d'observation d'une expérience : c'est le temps accordé au système pour réagir avant de considérer que celui-ci ne réagira plus. Il est conseillé d'effectuer un ajustement de la fenêtre d'observation en fonction du temps d'exécution de la charge de travail déterminé lors d'expériences préliminaires. Les blocages noyau en opération surviennent généralement lors de l'exécution d'une boucle infinie en mode noyau ou l'attente d'une interruption lorsqu'elles ne sont plus autorisées. De tels comportements ne peuvent pas être observés par le système et exigent ainsi la mise en place de mécanisme de surveillance externe.

La Figure 3-6 montre deux chronogrammes possibles du déroulement de deux expériences.

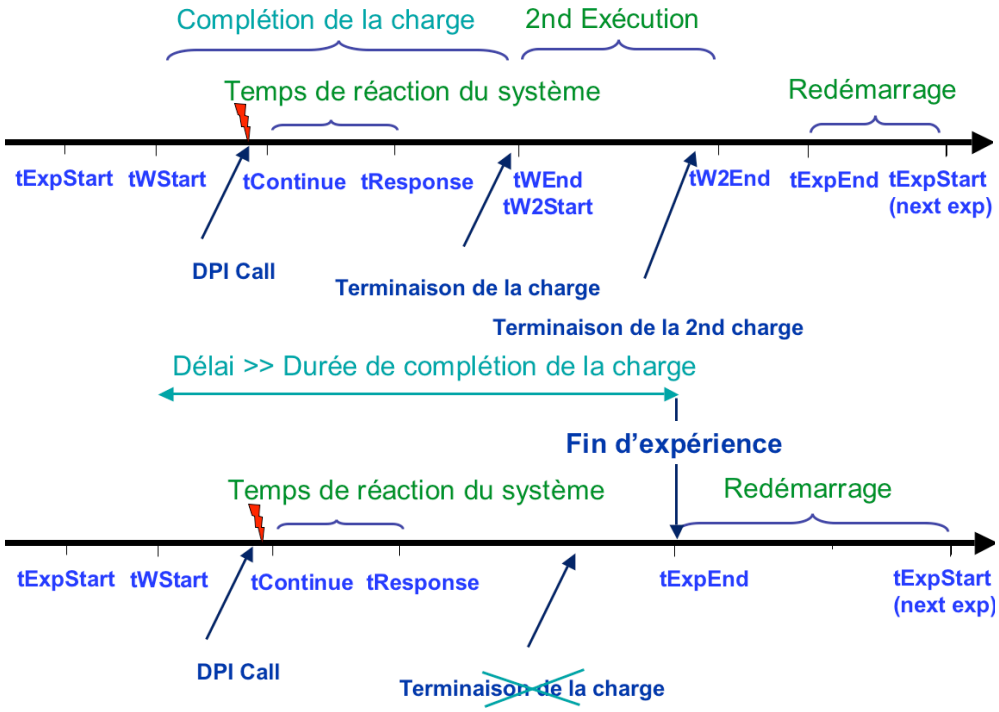


Figure 3-6 Chronogramme de deux expériences

L'instant *tExpStart* correspond au lancement de l'expérience, une fois le système redémarré dans un état stable. Lors de cette étape, les outils d'injection de fautes et d'observation du système sont insérés dans le système. Une fonction cible et une faute sont déterminées pour cette expérience. Durant la campagne, pour chaque fonction, tous les paramètres sont corrompus par les valeurs correspondantes de la charge de fautes (une faute par expérience). L'instant *tWStart* correspond au lancement de la charge de travail jusqu'à l'utilisation de la fonction cible à l'instant « *DPI Call* » par un pilote prenant en charge une requête applicative de la charge. L'exécution est alors interrompue pour effectuer la corruption. Un observateur sur le code retour d'erreur de la fonction est mis en place puis l'exécution de la fonction avec un paramètre corrompu reprend. Le retour du code d'erreur de la fonction, observation interne, est relevé lors de la terminaison de la fonction. La charge de travail reprend son exécution à *tContinue*.

La première observation externe est effectuée à l'instant *tResponse*. *tResponse - tContinue* correspond au temps de réaction du système. Une fois terminée la première exécution de la charge de travail en présence de la faute injectée à *tWEnd,* la seconde exécution est effectuée pour enrichir les observations et permettre d'affiner l'interprétation de l'expérience. Elle débute à *tW2Start* et se termine à *tW2End*. L'expérience se termine à *tExpEnd* par le redémarrage de la machine.

Le chronogramme de la partie inférieure présente le cas où la charge de travail ne se termine pas. Un temporisateur *TimeOut* est utilisé pour assurer le redémarrage de la machine lorsque l'expérience en cours dépasse significativement le délai prévu. Cette observation et le redémarrage de la machine ciblée sont effectués par une machine extérieure. En effet, le système ne peut pas observer son propre blocage.

3.4 P` ints de vue et interprétati` n

Les observations décrites dans le paragraphe précédent (paragraphe 3.3.3) offrent une base sur laquelle divers types d'analyses peuvent être effectués. Ces analyses permettent d'évaluer, vis-à-vis de ses besoins de sûreté de fonctionnement, l'impact sur le système d'une combinaison de comportements.

En pratique, différentes interprétations des mesures sont possibles en fonction du contexte spécifique dans lequel le noyau est testé (pour une future intégration dans le cas d'un COTS). En particulier, quand le système favorise un comportement correct de la charge de travail, alors les notifications d'erreurs par des codes retour d'erreurs ou quelquefois des gels du noyau peuvent être considérés comme des comportements corrects, ou du moins acceptables. Réciproquement, les comportements inacceptables correspondront, par exemple, à une interruption inopinée de certaines applications sans notifications d'erreurs au préalable. Inversement, cette catégorie sera tolérée dans une certaine mesure dans le cas d'un système à la recherche d'une grande disponibilité du système, y compris au détriment de certaines applications. La prise en compte du domaine d'application est abordée de manière détaillée dans [Duraes & Madeira 2002].

Ce problème est exacerbé dans le cas où plusieurs observations (un code retour d'erreur et un blocage noyau, par exemple) sont observées simultanément durant la même expérience d'injection de fautes. Alors, ces expériences doivent être analysées avec beaucoup de

précautions pour prendre en compte le point de vue de sûreté de fonctionnement dans lequel elles sont évaluées. La classification des manifestations de fautes est une première étape nécessaire pour permettre l'analyse tenant compte des points de vue de sûreté de fonctionnement.

3.4.1 Caractérisation des fautes

La caractérisation des fautes est un problème récurrent dans les études expérimentales d'injection de fautes. Ce paragraphe présente les manifestations typiques des défaillances qui peuvent se produire lors d'une campagne sur un système d'exploitation et les différentes stratégies qui peuvent être employées pour les analyser.

Soient *I* l'ensemble des expériences d'une campagne d'injection de fautes, et *A* l'ensemble d'expériences où la faute injectée est activée. Soient *F* et *D* les ensembles d'expériences où l'on observe respectivement une défaillance ou une détection d'erreurs. Le diagramme de Venn de la Figure 3-7 illustre ces ensembles.

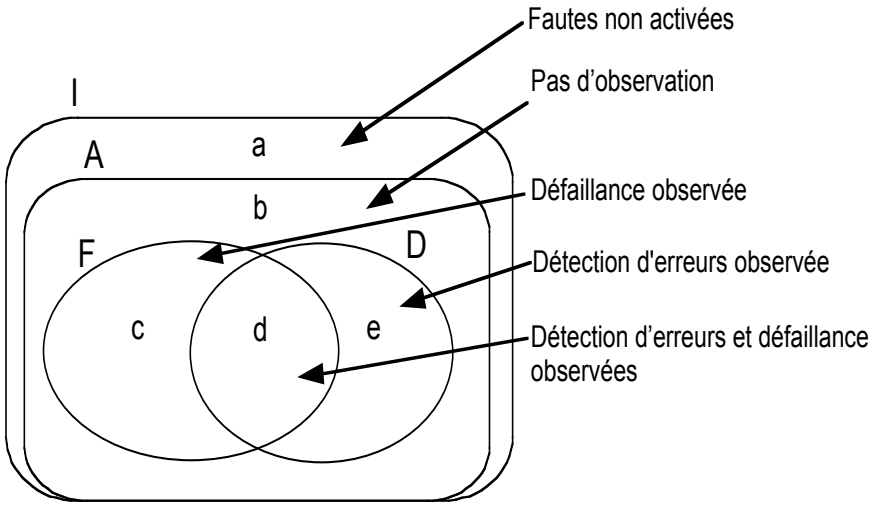


Figure 3-7 Manifestations de fautes

Les défaillances *F* sont provoquées par les erreurs qui se sont propagées à l'application ou au noyau, les faisant échouer en terme de valeur ou de terminaison.

Concernant la détection des erreurs (*D*), les mécanismes les plus communs de détection des erreurs (*Errors Detection Mecanisms* - EDM) fournis par un noyau sont les alarmes (surtout dans le cadre de noyaux temps reél), le code retour d'erreur et les exceptions (voir, par exemple, le Tableau 3-4).

Nous établissont une classification des expériences selon les manifestations de faute observées dans une campagne. Les expériences favorables à la classification sont les suivantes :

- Aucune manifestation n'est observée pendant l'expérience d'injection de fautes. Si la zone corrompue par la faute n'est pas employée par le processeur, la faute n'est pas activée (partition *a*). Réciproquement, si malgré l'activation de la faute, les résultats

fournis par l'application sont corrects, nous concluons que l'erreur correspondante a été intrinsèquement tolérée par le système (partition *b*). C'est aussi le cas si aucune erreur n'est observée en raison d'une latence élevée de l'erreur ou si l'erreur a été masquée.

- Une manifestation simple est observée pendant l'expérience d'injection de faute. Par conséquent, nous concluons que le système a échoué (partition *c*) ou qu'un EDM a signalé une erreur (partition *e*).

Considérons les expériences où une défaillance et une détection des erreurs sont observées (partition *d*). L'interprétation des expériences où l'on observe les occurrences de défaillances et de détections d'erreurs dépend de manière significative de la stratégie employée pour analyser cette partition.

- En général, signaler une erreur après une défaillance qui s'est déjà produite présente peu d'intérêt. L'ordre entre les événements est alors important. Dans ce contexte, une expérience dans la partition *d* est classifiée comme positive (détection d'erreur) si on observe l'activation d'un EDM en premier événement. Réciproquement, quand une défaillance se produit avant n'importe quelle détection d'erreurs, le système a échoué.
- Néanmoins, on peut considérer positivement le fait que le système fournit un EDM dans une fenêtre temporelle donnée. En effet, on peut considérer que la fenêtre temporelle correspondant à la durée de l'expérience est très petite, et donc l'ordre entre les événements n'est plus une information essentielle [Chevochot & Puaut 2001]. Dans ce cas, une expérience dans la partition *d* est systématiquement classifiée comme une détection d'erreur.
- D'un point de vue plus pessimiste, on peut considérer que les mécanismes de détection des erreurs fournis par certains systèmes sont de peu d'utilité quand une défaillance se produit [Marsden *et* Fabre 2001]. Des expériences dans la partition *d* sont alors classifiées comme défaillance.

Nous allons illustrer dans les deux prochains paragraphes le résultat de l'application de cette classification sur les observations effectuées lors de nos travaux sur deux outils différents d'injection de fautes.

3.4.2 Application à MAFALDA-RT

Cible	Délai dépassé	Résultats incorrects	Blocage applicatif	Blocage système	Alarme	Statut d'erreur	Exception	Pas d'obs.
mSCH	1	21	62	1	6	3	783	405
mTIM	1	44	22	0	5	53	680	541
pTIM	0	24	47	4	52	286	0	1068

Tableau 3-5Distribution des manifestations de fautes

Le Tableau 3-5 présente la distribution des manifestations des fautes observées lors de trois campagnes menées avec l'outil MAFALDA-RT [Rodriguez *et al.* 2002].

Lors de ces campagnes, des fautes de type « inversion de bit » ont été injectés dans les cibles suivantes (voir paragraphe 2.4):

- Le segment de code correspondant au module d'ordonnancement du micronoyau (campagne *mSCH*),
- Le segment de code correspondant au module de gestion des temporisateurs du micronoyau (campagne *mTIM*),
- Les paramètres des « appels système » vers le module de gestion des temporisateurs (campagne *pTIM*).

La faute injectée est déterminée aléatoirement selon 3 composantes : le paramètre à corrompre, le numéro de bit à inverser et l'instant d'injection. Les résultats présentés prennent en compte l'ordre d'occurrence des manifestations (voir paragraphe précédent). Seules les expériences où les fautes ont été effectivement activées, l'ensemble *A* (*I-a*) dans le paragraphe précédent, sont représentées.

L'ensemble *F* des défaillances correspond aux observations de dépassements d'échéance (*Deadline Missed*), blocages d'application (*Application Hang)*, blocages du système (*System Hang*) ou résultats incorrects (*Incorrect Results*). Les résultats de la campagne *mSCH* révèlent que 6,6% (rapport de *F* sur *A*) des fautes injectées et activées se sont propagées puis ont provoqué une défaillance. L'absence dans l'interface API de fonctions liées au composant d'ordonnancement du micronoyau entraîne un ensemble restreint d'observation de statuts d'erreur. L'ensemble *D* (regroupant les événements *alarme*, *statut d'erreur* et *exception*) représente 61,8% (rapport de *D* sur *A*) des expériences, composées donc en grande partie d'exceptions. Enfin, dans un tiers des cas (31,6%) les expériences menées fournissent le résultat attendu (ensemble *b*), en termes de valeur et de synchronisation, en dépit de l'activation effective de la faute injectée.

À l'inverse, la campagne *mTIM* révèle les défaillances non détectées -*F*- correspondent à 54,8% des expériences où la faute a été injectée et activée. Cependant, la diversité des EDM et supérieure par rapport à la campagne *mSCH*. Ce résultat s'explique par la présence d'une API dans le module de gestion des temporisateurs, permettant l'observation d'un nombre conséquent de statut d'erreur. Cependant, à l'image de la campagne *mSCH*, la majorité -50,5%- (correspondant au rapport de *D* sur *A*) des inversions de bits effectuées durant la campagne *mTIM* ont été détectées par les EDM.

Concernant la campagne *pTIM*, la principale différence avec les campagnes précédentes est l'absence d'activation des mécanismes d'exceptions. Ce comportement favorise l'occurrence des événements correspondant à des alarmes (3,5%) et à des statuts d'erreur (19,3%) dans *D*. En effet, les exceptions sont davantage liées aux erreurs de niveau bas (par exemple, les fautes de segmentation) qu'aux erreurs affectant les paramètres des appels système, à la différence du mécanisme de statut d'erreur, prévu pour vérifier la validité des appels de système. D'autre part, certains des paramètres corrompus ont modifié la fréquence d'activation de certaines tâches, menant à un plus grand nombre d'alarmes. L'impact des fautes injectées lors de cette

campagne semble inférieur comme le montre le taux particulièrement élevé d'expériences correctes (72%, rapport de *b* sur *A*).

Cependant, comme nous l'avons indiqué au paragraphe précédent, les taux de défaillance -*F*- et de détection des fautes -*D*- peuvent évoluer de manière significative selon la stratégie d'analyse des événements observés, en particulier dans le cas où on observe une combinaison de ceux-ci. Les résultats présentés sur le Tableau 3-5 tiennent compte de l'ordre de l'occurrence de tels événements.

Cependant, si nous considérons que l'ordre des événements n'est pas primordial, les résultats peuvent être présentés différemment, accordant la priorité aux événements de détection des erreurs ou aux événements d'échec. La figure 3-8 présente cette analyse. Elle montre des taux de défaillances pour les campagnes précédentes selon les trois stratégies différentes d'analyse; premier événement (ordre des événements considérés, utilisée pour le Tableau 3-5), priorité à D (priorité aux événements de détection des erreurs), et priorité à F (priorité aux événements correspondant à des défaillances).

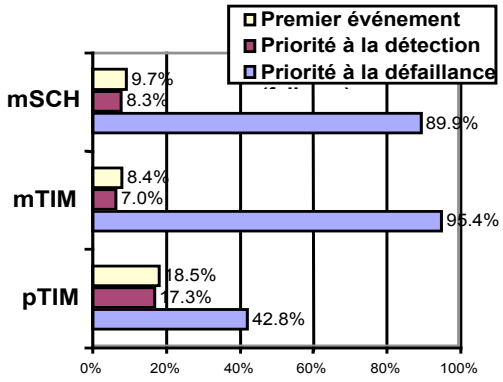


Figure 3-8 Taux de défaillances selon la stratégie considérée

Le taux de défaillance -rapport de *F* sur *A*- inclut les résultats correspondants à des dépassements de délai, des blocages d'application, des blocages du système et l'observation de résultats incorrects fournis par le système. Son complément correspond à l'assurance d'une détection des erreurs par un mécanisme du système : Il comprend les alarmes, les statuts d'erreur et les exceptions. La catégorie correspondant aux résultats corrects n'est pas représentée.

La campagne *mSCH* révèle que, dans 98,3% des cas, une détection d'erreur a précédé une défaillance. Ce taux est similaire pour la campagne *mTIM* (98,4%) et inférieur lors de la campagne *pTIM* (95,3%). Les stratégies « premier événement » et « priorité à la détection » fournissent des taux de défaillances similaires. L'origine provient du fait que, lors de l'occurrence des événements de détection d'erreurs et de défaillance, la détection d'erreur est le premier observé durant nos campagnes. Réciproquement, quand la stratégie « priorité à défaillance » est utilisée, les résultats différant totalement et révèle des taux de défaillances considérablement plus élevés. En effet, les nombreuses expériences (80,2% pour *mSCH*, de 87% pour *mTIM* et 24,3% pour *pTIM*) où une défaillance a été précédée par une détection d'erreur sont maintenant incluses dans les chiffres correspondant à une défaillance. Alors nous

pouvons observer que seule la campagne *pTIM* est moins affectée par la stratégie d'analyse utilisée du fait de son relativement faible nombre d'expériences avec des événements combinés de détection d'erreurs et de défaillance (seulement 25,5% du total).

3.4.3 Application à notre méthode

Le Tableau 3-6 fournit un essai pour caractériser les observations de l'outil d'injection de fautes sur la DPI selon la classification des manifestations de fautes présentée dans le paragraphe 3.4.1. Le premier ensemble de colonnes montre les combinaisons possibles des observations définies dans le Tableau 3-3.

	Observations					**Priorité à**		
#	**Notifications**		**Défaillances**			**Premier**	**Erreur**	**Défaillances**
	EC	**XC**	**WA**	**WI**	**KH**	**evt**	**Notifiée**	
O1	1	0	0	0	0	EC	EC	EC
O2	1	1	0	0	0	EC	EC+XC	EC+XC
O3	0	1	0	0	0	XC	XC	XC
O4	1	1	0	0	1	EC	EC+XC	KH
O5	1	0	0	0	1	EC	EC	KH
O6	0	1	0	0	1	XC	XC	KH
O7	0	0	0	0	1	KH	KH	KH
O8	1	1	1	X	1	EC	EC+XC	KH+WA
O9	1	0	1	X	1	EC	EC	KH+WA
O10	0	1	1	X	1	XC	XC	KH+WA
O11	0	0	1	X	1	KH	KH+WA	KH+WA
O12	0	0	0	0	0	*No Obs.*	*No Obs.*	*No Obs.*
O13	1	1	1	X	0	EC	EC+XC	WA
O14	1	0	1	X	0	EC	EC	WA
O15	0	1	1	X	0	XC	XC	WA
O16	0	0	1	X	0	WA	WA	WA
O17	1	1	0	1	0	EC	EC+XC	WI
O18	1	0	0	1	0	EC	EC	WI
O19	0	1	0	1	0	XC	XC	WI
O20	0	0	0	1	0	WI	WI	WI
O21	1	1	0	1	1	EC	EC+XC	WI + KH
O22	1	0	0	1	1	EC	EC	WI + KH
O23	0	1	0	1	1	XC	XC	WI + KH
O24	0	0	0	1	1	WI	WI + KH	WI + KH

Tableau 3-6 Résultats et diagnostics

Deux catégories sont distinguées :

- Les notifications d'événements (erreur explicitement rapportée) comptent les observations au niveau interne (Code d'erreur - *EC*) et au niveau externe (Exceptions - *XC*);
- Les modes de défaillance sont liés à l'état de la charge applicative (Résultat incorrect - *WI* ou interruption - *WA* de la charge de travail) ou du noyau (Blocage noyau- KH).

La deuxième partie de la table illustre les diagnostics selon un ensemble de critères simples lorsque plusieurs événements différents sont relevés lors de la même expérience :

- L'ordre de l'occurrence des événements observés est respecté.
- La priorité est accordée à la notification d'erreur.
- La priorité est accordée aux modes de défaillance.

En premier lieu, il faut remarquer que tous les événements considérés ne sont pas entièrement indépendants et, en conséquence, toutes les combinaisons ne sont pas possibles. En particulier, c'est le cas pour les modes de défaillance correspondant à l'arrêt de la charge de travail (*WA*) et à une terminaison incorrecte de cette charge (*WI*) : en effet, l'observation *WA* décide en partie du résultat de l'observation *WI*, c'est-à-dire, on ne peut observer aucun *WI* quand un *WA* a été diagnostiqué. Le résultat n'ayant pas été fourni par la charge de travail, on ne peut juger sa pertinence et de sa véracité avec le résultat attendu fourni par l'oracle (voir paragraphe 3.3.2). Cette dépendance est identifiée par le symbole "*X*" dans le Tableau 3-6 et explique la taille réduite du tableau (24 lignes au lieu de 32).

On observe que la ligne *O12* est particulière. Elle indique des cas lorsque aucune observation n'a pu être effectuée. Leurs interprétations dépendent étroitement du contexte spécifique où l'analyse est conduite. Nous verrons dans le chapitre suivant que ces résultats peuvent être comptés positivement ou négativement selon le cas.

Quand plusieurs événements sont observés lors de la même expérience, il est possible de classer les résultats par catégorie. L'approche habituelle est d'accorder la priorité au premier événement observé. Cependant, il n'est pas toujours possible d'avoir des mesures temporelles précises pour tous les événements. En effet, elles exigent habituellement une instrumentation supplémentaire et un outil plus sophistiqué. Ces exigences peuvent être une difficulté supplémentaire pour la portabilité d'un étalon de sûreté de fonctionnement. D'autres solutions proposent d'accorder la priorité :

- Aux notifications d'erreur (les codes retour d'erreur et les exceptions), correspondant à la priorité à la détection du paragraphe précédent.
- Aux modes de défaillance à observer (arrêt de charge de travail, charge de travail incorrecte et blocage du noyau), correspondant à la priorité à la défaillance dans MAFALDA-RT.

Comme nous l'avons dit précédemment, on peut clairement voir une différence entre les deux : la première est, à priori, optimiste tandis que la seconde est plutôt pessimiste quant aux capacités du système à tolérer la faute. Dans les deux cas, si des événements multiples sont

observés, ceux ne concernant pas la catégorie prioritaire sont toutefois enregistrés pour des analyses supplémentaires plus précises. Le tableau 3.6 montre également le traitement de la chronologie des événements appartenant à une même catégorie. Par exemple, quand la priorité est accordée aux modes de défaillance, *WA+KH* signifie que nous avons observé un arrêt de la charge de travail suivi d'un blocage du noyau. En raison de la manière dont les événements considérés, sont rassemblés dans les deux stratégies "Priorité à la notification d'erreur" et "premier événement", les deux stratégies sont très proches. En effet, les notifications précèdent généralement les modes de défaillance considérés. C'est effectivement le cas dans notre campagne d'injection de fautes.

3.4.4 Prise en compte du point de vue

À partir de la caractérisation de l'impact d'une faute, on peut définir des interprétations plus opérationnelles grâce à l'ajout de différents points de vue prenant en compte des mesures orientées sûreté de fonctionnement recherchées par l'utilisateur de l'étalon. En effet, nous considérerons trois interprétations qui correspondent à trois contextes distincts ayant pour priorités :

- La notification d'événements par le noyau (*Responsiveness RK*).
- La sécurité (innocuité) au niveau de la charge de travail (*Safety SW*).
- La disponibilité du noyau (*Availability AK*).

Le principal intérêt de l'interprétation associée à la notification d'événements par le noyau est de considérer comme positives toutes les notifications d'erreurs du système. Les expériences où on relève des modes de défaillances et des notifications d'erreur sont considérées comme une réaction positive du système en présence de fautes. Le fait que le système ait réagi par une notification d'erreurs peut être considéré positivement même si des défaillances sont apparues. Les cas négatifs représentent l'absence de réaction du système, indiquant que la faute a bien eu un impact sur le système qui n'a pas réagi de manière explicite.

La mesure associée à la sécurité au niveau de la charge de travail correspond au cas où le résultat obtenu reste conforme au résultat attendu, en présence de fautes. En conséquence, la plupart des résultats favorables à cette mesure correspondent aux événements induisant un comportement fiable du résultat ou silencieux. Les notifications d'erreurs et les blocages du noyau correspondent alors à un comportement considéré plus favorable qu'un arrêt de la charge de travail, ou un accomplissement incorrect au niveau applicatif.

À l'inverse, l'interprétation associée à la mesure de disponibilité du noyau classera différemment les expériences si elles présentent l'occurrence d'un blocage noyau ou une interruption de la charge de travail. En effet, un blocage noyau constitue l'événement ayant le plus important impact sur la disponibilité. Il nécessite le plus souvent une intervention humaine et entraîne la mise en place de procédés pour assurer l'intégrité du système (entre autres, le système de fichiers). Au contraire, une interruption de la charge de travail n'a pas nécessairement d'impact sur les autres tâches et ne nécessite pas une intervention lourde de l'équipe de maintenance.

Le Tableau 3-7 montre comment ces mesures de sûreté de fonctionnement peuvent être liées aux observations présentées dans le Tableau 3-6.

Point de vue : Notification d'événements par le système

	#	Cas [-O12]	Explications
+	**RK1**	O1-O3	Une notification d'erreur est relevée avant la terminaison correcte de la charge de travail.
+	**RK2**	O4-O6, O8-O10, O13-O15, O17-O19, O21-O23	Une notification d'erreur est relevée avant l'occurrence d'une défaillance.
-	**RK3**	O16	Pas de notification d'erreur et la charge de travail est interrompue.
-	**RK4**	O7, O11, O24	Pas de notification d'erreur et un blocage noyau a été observé.
-	**RK5**	O20	Pas de notification d'erreur et la charge de travail a fourni des résultats incorrects.

Point de vue : Sécurité au niveau de la charge de travail

	#	Cas [-O12]	Explications
+	**SW1**	O1-O3	Une notification d'erreur est relevée. La charge de travail s'est terminée correctement.
+	**SW2**	O4-O7	La charge de travail s'est terminée correctement puis un blocage noyau a été observé.
+	**SW3**	O8-O11, O13-O16	La charge de travail a été interrompue puis un blocage noyau a été observé.
-	**SW4**	O21-O24	La charge de travail a fourni des résultats incorrects puis un blocage noyau a été observé.
-	**SW5**	O17-O20	La charge de travail a fourni des résultats incorrects sans être suivi d'un blocage noyau.

Point de vue : Disponibilité du noyau

	#	Cas [-O12]	Explications
+	**AK1**	O1-O3	Une notification d'erreur est relevée avant la terminaison correcte de la charge de travail.
+	**AK2**	O13-O20	La charge de travail a été interrompue ou a fourni un résultat incorrect.
-	**AK3**	O4-O7	Un blocage noyau a été observé et la charge de travail s'est terminée correctement.
-	**AK4**	O8-O11, O21-O24	Un blocage noyau a été observé et la charge de travail a été interrompue ou s'est terminée de manière incorrecte.

Tableau 3-7 Mesures de sûreté de fonctionnement

Afin d'expliciter l'exploitation, les résultats différents d'une expérience à l'autre peuvent être regroupés dans des catégories équivalentes de gravité selon une mesure spécifique. Chaque catégorie caractérise un niveau de sûreté de fonctionnement en respect des critères de l'interprétation. Les catégories sont classées selon un niveau croissant de sévérité : L'index indique les cas les plus favorables. Nous avons mis en évidence par l'intermédiaire des signes « + » et « - » les résultats positifs et négatifs dans la mesure choisie. Par exemple, la catégorie *AK1* respecte le niveau le plus élevé de la disponibilité du noyau. La catégorie *AK2* respecte les critères de disponibilité du noyau. Inversement, les catégories AK3 et AK4 ne respectent pas

ces critères, elles sont donc notées par un signe « - » dans le tableau. Nous recommandons ce pendant de garder l'ensemble des informations des résultats d'expérience pour permettre éventuellement une interprétation plus fine et personnalisée (les catégories AK3 et AK4 n'ont pas le même niveau de gravité en termes de disponibilité du noyau).

Comme nous l'avons évoqué dans le paragraphe précédent, la ligne *012* est particulièrement difficile à interpréter hors de son contexte. Nous avons donc volontairement exclu cette catégorie d'observation de l'analyse des résultats avec prise en compte des points de vue de sûreté de fonctionnement. Nous y reviendrons au cours du chapitre suivant.

Le tableau présente également deux positions pour définir les résultats d'une expérience. Elles dépendent du niveau de connaissance de l'observateur. Si tous les résultats sont considérés ou si uniquement le point de vue de l'utilisateur est pris en compte. La colonne de droite donne une description complète et précise définissant chaque catégorie d'une mesure.

3.5 C`nclusi`n

L'impact des défaillances des pilotes de périphériques sur la sûreté de fonctionnement d'un noyau de système est méconnu. Les systèmes d'exploitation les plus répandus présentent une approche similaire d'intégration des pilotes de périphériques au système en opération:

- Ils sont situés dans le même espace d'adressage que le noyau.
- Ils bénéficient des privilèges d'exécution maximaux du système.
- Ils utilisent une interface réservée pour communiquer entre eux et avec le noyau.

Cette interface est composée de fonctions similaires d'un système à l'autre, seule leur implémentation évolue. À partir de ce constat, elle constitue une opportunité exceptionnelle pour l'évaluation du comportement du système en présence de pilotes défaillants. Nous proposons d'effectuer une injection de fautes dans les paramètres d'une fonction appelé par un pilote. L'injection a lieu durant l'exécution de la fonction. Dans l'objectif d'obtenir une technique reconnue, nous avons utilisé un modèle de fautes correspondant à la corruption de paramètres lors de ces appels privilégiés. Ce modèle de fautes à déjà fait preuves d'études avancées lors de précédents travaux.

Nous avons vu que la caractérisation du système demande la mise en place de mécanismes d'observations d'une charge à la fois « acteur » (Elle active les fautes) et « témoin » (Elle témoigne de leurs effets). L'observation se réalise à deux niveaux, applicatif et système par l'analyse du déroulement de la charge de travail et des mécanismes de détection de fautes. Cependant, les difficultés d'interprétation de ces observations nous a amené à définir un cadre global prenant en compte deux facteurs :

- Classification des observations selon le domaine
- Prise en compte des besoins de sûreté de fonctionnement.

La première a permis la résolution des observations multiples et des difficultés de l'interprétation des résultats fournis par une plateforme expérimentale d'injection de faute. La

seconde a permis d'évaluer le composant étalonné en fonction de niveau de sûreté de fonctionnement correspondant à l'attente du système destiné à intégrer le composant.

- La notification d'événements par le système.
- La sécurité au niveau de la charge de travail.
- La disponibilité du noyau.

Suite à la présentation dans ce chapitre de la méthode proposée pour l'évaluation du comportement d'un système d'exploitation vis-à-vis de pilotes défaillants, nous allons voir dans le chapitre suivant l'implémentation de cette méthode lors de la mise en œuvre d'une campagne d'expériences pour l'évaluation du noyau *Linux*.

Chapitre 4 Mise en œuvre expérimentale et résultats

La méthode présentée dans le chapitre précédent permet d'évaluer la robustesse de l'interface d'un noyau de système d'exploitation vis-à-vis d'un pilote défaillant. Pour la développer et la valider, nous avons mis en œuvre une plate-forme expérimentale dont l'objectif est de réaliser des campagnes d'expériences sur un système connu du grand public. Le choix de *Linux* a été motivé par plusieurs critères. En premier lieu, ce système peut être obtenu dans sa version complète gratuitement sur Internet, avec une documentation riche et aisément accessible. Il représente une part de plus en plus importante des nouveaux systèmes d'exploitation personnels et professionnels installés. Ensuite, le code source de *Linux* est libre d'accès, ce qui permet une analyse plus facile des effets du prototype d'injection de fautes. La possibilité de consulter le code du système favorise l'élaboration et la validation de l'outil. Enfin, *Linux* bénéficie d'une communauté très active.

Pour garantir la validité de la technique proposée dans ce manuscrit, nous avons également exploré sa mise en place sur un produit différent, *Windows XP*. Nous présentons tout au long de ce chapitre les moyens pour mettre en œuvre la technique sur le produit de *Microsoft*. En effet, la comparaison des résultats obtenus sur les deux systèmes reste un des principaux objectifs de l'étalonnage de la sûreté de fonctionnement et une perspective essentielle de nos travaux.

Dans la première partie de ce chapitre, nous décrivons la composition de l'interface entre les pilotes de périphériques et le noyau *Linux*. Nous examinons le contenu de cette interface dans le cadre d'un test de robustesse. Nous précisons ensuite le profil d'exécution nécessaire à l'activation du système et de sa DPI à travers la composition logicielle de sa charge de fautes et son activité. Nous détaillons alors la plate-forme expérimentale et les techniques d'observation et de mesure du système. Nous présentons les outils logiciels développés et nous décrivons la conduite d'une campagne d'expériences. La dernière partie de ce chapitre présente les résultats obtenus et les interprétations que l'on peut effectuer en respect de divers points de vue et des différentes techniques d'analyses.

4.1 L'interface pil` te n` yau de *Linux*

La caractérisation d'un système par un test de robustesse sur son interface pilote noyau (DPI) demande une étude préalable sur le système cible. En effet, il est essentiel d'identifier les fonctions qui composent l'interface. Nous décrivons dans ce paragraphe les types de paramètres présents dans les appels de fonction à la DPI et nous en déduisons les valeurs corrompues à injecter.

4.1.1 Les symboles de *Linux*

Nous avons vu dans le premier chapitre que les symboles de *Linux* forment l'interface de programmation entre les extensions du noyau et celui-ci. Les fonctions des différents services présentés dans le chapitre 3 sont détaillées dans le Tableau 4-1.

Services	Symboles associés
Interface	setup_arg_pages, copy_strings_kernel, do_execve, flush_old_exec, kernel_read, open_exec
Configuration	register_chrdev, unregister_chrdev, register_blkdev, unregister_blkdev, tty_register_driver, tty_unregister_driver, tty_std_termios,
Ressource Allocation	request_dma, free_dma, dma_spin_lock, request_resource, release_resource, allocate_resource, check_resource, __request_region, __check_region, __release_region, ioport_resource, iomem_resource,
Error Handling	N/A
Buffer management	mark_buffer_dirty, set_buffer_async_io, __mark_buffer_dirty, __invalidate_buffers, unlock_buffer, __wait_on_buffer, generic_buffer_fdatasync, init_buffer, refile_buffer, tty_flip_buffer_push, fsync_buffers_list, buffer_insert_inode_queue, buffer_insert_inode_data_queue,
Inter-module Communication	inter_module_register, inter_module_unregister, inter_module_get, inter_module_get_request, inter_module_put, try_inc_mod_count,
System I/O	do_mmap_pgoff, do_munmap, do_brk, exit_mm, exit_files, exit_fs, exit_sighand,
Control Block management	blksize_size, hardsect_size, blk_size, blk_dev, is_read_only, set_device_ro, bmap, sync_dev, devfs_register_partitions, blkdev_open, blkdev_get, blkdev_put, ioctl_by_bdev, grok_partitions, register_disk, tq_disk, init_buffer, refile_buffer, max_sectors, max_readahead,
Memory management	_alloc_pages, __alloc_pages, alloc_pages_node, __get_free_pages, get_zeroed_page, __free_pages, free_pages, num_physpages, kmem_find_general_cachep, kmem_cache_create, kmem_cache_destroy, kmem_cache_shrink, kmem_cache_alloc, kmem_cache_free, kmem_cache_size, kmalloc, kfree, vfree, __vmalloc, vmalloc_to_page, mem_map, remap_page_range, max_mapnr, high_memory, vmtruncate, find_vma, get_unmapped_area, init_mm,
Interrupt management	request_irq, free_irq, irq_stat
Time management	add_timer, del_timer, add_wait_queue, _wait_queue, finish_wait,

Tableau 4-1 Symboles composant les services fournis à travers la DPI Linux

Dans notre approche, seules les fonctions provenant du noyau minimal de *Linux* sont ciblées. Elles concernent les services essentiels à la bonne marche d'un système d'exploitation et constituent le noyau cible, à la différence du système étendu.

Chaque ensemble de symboles est dédié à une tâche précise dans le système : par exemple, pour la réservation d'espace mémoire, les fonctions *kmalloc, kfree* permettent respectivement la réservation et la libération d'un espace mémoire dans le noyau. Certains symboles ont des noms très proches (*_alloc_pages* et *__alloc_pages,* par exemple) pour différencier une même fonction avec deux possibilités d'appel (l'équivalent de la « multisignature »), permettant de fixer certains paramètres pour plus de facilité d'utilisation. La programmation des pilotes de périphériques sous *Linux* ne permet pas une gestion d'erreur approfondie en collaboration avec le noyau, comme c'est le cas sous d'autres systèmes comme *Windows XP*. Ce manque constitue une des plus grandes difficultés à la programmation en mode noyau sous *Linux*. En effet, elle rend le déboguage particulièrement difficile et fastidieux. Les dernières versions de *Linux* tentent de palier cette lacune.

Chacune de ces fonctions dispose d'un ensemble de paramètres pour permettre la réalisation de leurs instructions. Nous allons voir dans le prochain paragraphe quels types de paramètres sont présents sous *Linux*.

4.1.2 Les types dans *Linux*

Le nombre de types présents dans le code de *Linux* est très important. En effet, aux types classiques connus dans tous les systèmes, s'ajoutent les types pointant sur des structures, augmentant significativement ce nombre. En effet, de nombreux types sont des pointeurs vers des structures. Ce type peut lui-même correspondre au champ d'une nouvelle structure et ainsi de suite. Dans notre cas, la définition des types présents dans le code de *Linux* pourrait être composée de cet ensemble de types. Toutefois, la complexité de la mise en œuvre de la technique d'injection de fautes sur la DPI augmente avec la diversité des types de paramètres considérés.

Nous avons donc choisi de définir un nombre réduit de classes de type de paramètres, chacune généralisant les possibilités de chaque type de paramètres. La définition des classes de types est essentielle pour l'obtention de résultats les plus expressifs et précis possibles. Pour la mise en œuvre expérimentale, nous avons privilégié un niveau d'abstraction élevé pour d'éventuelles comparaisons avec un autre système d'exploitation, essentielles dans un contexte d'étalonnage.

Les classes de types rencontrés sous *Linux* sont présentées dans le Tableau 4-2. Certaines classes, comme la classe *mode,* permettent de faciliter l'analyse des comportements du système en présence de fautes. Les éléments de la classe de type Mode sont représentés sur quatre octets, cependant leur sémantique diffère de manière importante avec les autres paramètres (par exemple à la classe des entiers longs ou des identificateurs de processus). On peut alors relier les comportements anormaux du système en fonction de la sémantique du paramètre corrompu.

La classe correspondant aux pointeurs vers les structures permet de réduire l'analyse des paramètres liés aux structures du système. N'importe quel pointeur vers une structure est considéré comme appartenant à cette classe. Toutefois, il faut noter que la précision concernant le contenu du pointeur est perdue, ce qui ne permettra pas d'effectuer une analyse très fine des conséquences de la faute. En effet, on considérera ici de manière indifférente un pointeur de

structure simple d'un pointeur de structure complexe. Les autres classes de types sont communes et se retrouvent généralement dans le code des systèmes d'exploitation modernes.

Classe de Type	Description
Pointeur lecture/écriture	Cette classe regroupe les types de données qui représentent des pointeurs. C'est-à-dire la valeur dans l'adresse pointée qui sera lue ou écrite.
Pointeur vers structure	Les structures de données forment cette classe de types de données. Il s'agit de pointeurs qui pointent sur des structures simples ou complexes.
Pointeur temporel	Les structures de données représentant une quantité de temps sont représentées par le type (*timespec* *) et forment cette classe.
Entier signé	Cette classe représente tous les paramètres de type entier (*int*) qui ne sont pas inclus dans les classes ci-après.
Entier non signé	Cette classe représente les entiers non signés (*unsigned int*).
Identificateur de processus	Les processus sont identifiés par le noyau par un numéro unique. Dans les systèmes *Unix*, cet identificateur est généralement noté PID. Ils sont représentés par des entiers.
Descripteur de fichiers	Un processus identifie les fichiers qu'il manipule par des descripteurs de fichiers (noté *fd* dans les systèmes *Unix*). Il s'agit d'entiers caractérisant d'une manière unique un fichier ouvert. Chaque processus attribue un et un seul descripteur de fichier pour un fichier ouvert. C'est-à-dire, deux processus peuvent allouer deux descripteurs de fichiers différents pour le même fichier.
Mode	Ces types de données sont représentés sur quatre octets et permettent de définir les modes de lecture et d'écriture ainsi que les permissions d'accès. L'appel système *mmap*, par exemple, permet d'ajouter une zone mémoire dans l'espace d'adressage d'un processus. Cette zone mémoire possède une protection qui est définie à travers la mise à un ou à zéro d'un bit du quatrième paramètre de l'appel.

Tableau 4-2 Les classes de types de paramètre dans le code de Linux

Il est important de noter que les éléments de chaque classe ont un intervalle de définition commun. Nous précisons ces intervalles dans le prochain paragraphe. Le nombre de classes peut augmenter de manière très importante en fonction de l'analyse effectuée sur les pointeurs et surtout les structures pointées du système. En effet, on peut considérer que deux pointeurs de structures complexes différentes appartiennent à deux classes distinctes. Cette distinction permet de considérer les valeurs des champs d'un pointeur plutôt que la valeur du pointeur lui-même. L'étape préliminaire d'analyse de la composition de l'interface DPI d'un système est donc une étape cruciale dans l'évaluation du comportement d'un système en présence de pilotes défaillants.

4.1.3 Intervalles de validité des classes de paramètres

La particularité des classes de types présentées dans le paragraphe précédent réside dans leur intervalle de validité. Tous les types appartenant à une même classe disposent d'un intervalle de validité identique. Tout type peut être inclus dans au moins une classe. Cet intervalle de validité permet de déterminer les corruptions à effectuer lors de l'évaluation de la robustesse de la DPI.

Le Tableau 4-3 présente les bornes et les valeurs exceptionnelles des intervalles de validité des classes de types des paramètres de *Linux* conformément au principe identifié au chapitre 3.

Classe	Borne minimale	Borne maximale	Exceptionnelle
Pointeur lecture/écriture	*All bits = 0 (0x00000000)*	*All bits = 1 (0xFFFFFFFF*	*NULL*
Pointeur vers structure	*All bits = 0 (0x00000000)*	*All bits = 1 (0xFFFFFFFF*	*NULL*
Pointeur temporel	*All bits = 0 (0x00000000)*	*All bits = 1 (0xFFFFFFFF*	*NULL*
Entier signé	*INT_MIN(0x80000000)*	*INT_MAX(0x7FFFFFFF)*	*0*
Entier non signé	*0*	*ULONG_MAX(0xFFFFFFFF)*	*Aucune*
court non signé	*0*	*USHRT_MAX(0xFFFF))*	*Aucune*
Identificateur de processus	*INT_MIN(0x80000000)*	*INT_MAX(0x7FFFFFFF)*	*0*
Descripteur de fichiers	*INT_MIN(0x80000000)*	*INT_MAX(0x7FFFFFFF)*	*0*
Mode	*INT_MIN(0x80000000)*	*INT_MAX(0x7FFFFFFF)*	*0*

Tableau 4-3 Intervalles de validité des classes de type

La borne minimale représente le plus petit élément sémantique pouvant être codé sur le nombre d'octet alloué au type. Par exemple, un entier court non signé est codé sur deux octets. Le plus petit élément de cette classe a une valeur hexadécimale de *0x0000* et une valeur maximale de *0xFFFF*. De la même manière, un entier long signé est codé sur quatre octets, le plus petit élément est codé *0x80000000*, équivalent à *-2147483648* et le plus long *0x7FFFFFFF*, la valeur équivalente en positif à laquelle on enlève un, valeur réservée pour le *zéro*.

Zéro correspond à une valeur exceptionnelle de ces types. Nous la considérerons durant nos expériences pour les conséquences connues de l'utilisation inappropriée de la valeur *0*, en arithmétique par exemple. Il en est de même de la valeur *NULL* pour les pointeurs.

Certaines classes ont des intervalles de validité identiques. On notera que la classe des descripteurs de fichiers et la classe des identificateurs de processus disposent du même intervalle de validité. Ce phénomène est dû à la nature de leur représentation pour *Linux*. En

effet, ces classes sont toutes deux de type entier (*int*). Toutefois, les éléments de ces classes sont sémantiquement très différents. C'est pour conserver cette distinction sémantique que nous avons défini des classes différentes pour un même intervalle de validité.

4.1.4 Les types dans *Windows*

La plupart des systèmes d'exploitation sont développés avec le langage de programmation *C*, voire *C++* pour les récentes implémentations. Les types présents dans le code des noyaux sont donc similaires de l'un à l'autre. On distingue principalement cinq classes de types sous *Windows* :

- Les pointeurs
- Les entiers
- Les entiers non signés
- Les booléens
- Les chaînes de caractères

Ces classes peuvent être distinguées en sous-classes, comme la catégorie correspondant aux pointeurs, afin d'affiner leur domaine de définition. Les intervalles de validité de chacune de ces catégories sont sensiblement les mêmes que pour *Linux*. La principale distinction est le type booléen (0 à 0xFF) qui forme une nouvelle classe. Les travaux concernant l'outil d'injection de fautes Ballista présenté dans [Koopman *et al*. 1997] montrent comment les types de paramètres peuvent être distingués dans un système d'exploitation. On retrouve systématiquement des paramètres correspondant, par exemple, à une longueur, des droits d'écriture ou encore des identificateurs de fichiers

4.2 Plate-f^ rme expérimentale

Nous allons présenter au cours de ce paragraphe la plate-forme expérimentale ayant servi de support à la mise en œuvre de la technique d'évaluation du comportement d'un système en présence de pilotes défaillants. Nous décrirons dans un premier temps le banc d'essai. Nous verrons ensuite notre technique logicielle d'injection de fautes dans *Linux*. Nous présenterons ensuite le fonctionnement des outils permettant d'effectuer les observations et les mesures des machines accueillant les campagnes d'expériences.

4.2.1 Description générale

La Figure 4-1 décrit le banc d'essai supportant les campagnes d'expériences. Les expériences ont été effectuées sur une plate-forme comportant quatre machines équipées d'une architecture *Intel* et des processeurs *Pentium* de 133 mégahertz, chacune disposant d'une mémoire vive de 32 mégaoctets et de plusieurs périphériques classiques : un disque dur, un lecteur de disquette, un lecteur de disque compact, deux cartes réseau, une carte graphique, un clavier... Chacune des quatre machines est exploitée par la distribution *Debian 3.0* de *GNU/Linux*.

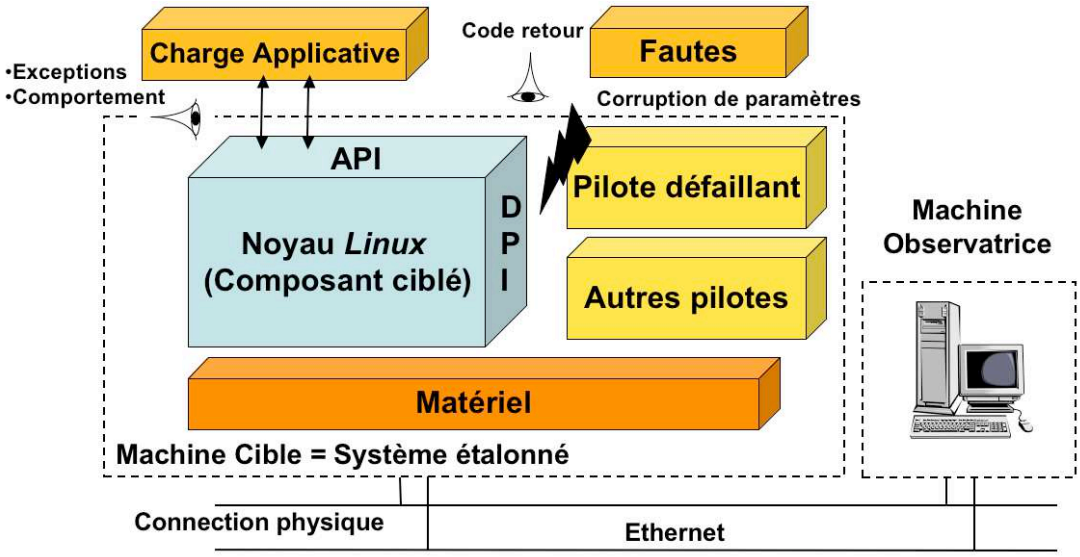


Figure 4-1 Vue d'ensemble du banc d'essais (une seule machine cible est représentée)

La machine cible soutient deux versions du noyau Linux : la 2.2.20 et la 2.4.18. L'utilisation de trois machines cibles en parallèle permet d'accélérer la conduite des expériences et renforce la confiance dans les observations effectuées. L'étude de deux noyaux différents permet d'évaluer et comparer l'évolution de la DPI d'un noyau à l'autre.

La quatrième machine (*Machine Observatrice*) est reliée aux machines cibles par l'intermédiaire d'un réseau Ethernet privé pour commander les expériences et pour fournir des moyens externes de surveillance des machines. En particulier, elle est indispensable pour observer et contrôler les machines cibles lorsque celles-ci sont bloquées après une expérience d'injection de faute. Pour réaliser ce contrôle quel que soit l'état de la machine ciblé, une connexion matérielle est nécessaire. Les autres observations (code retour, exceptions et comportement de la charge de travail) sont observées directement par la machine cible.

4.2.2 Spécifications du profil d'exécution

La détermination du profil d'exécution est une étape essentielle à la mise en œuvre expérimentale d'une technique d'étalonnage de la sûreté de fonctionnement. Nous avons vu dans le premier chapitre qu'il se compose de la charge de fautes et de l'activité :

- La charge de fautes représente l'ensemble des fautes qui vont être injectées durant la campagne d'expérience.
- L'activité ou "charge applicative de travail" est un ensemble d'applications simulant une activité "en opération" du système. Son rôle est de solliciter le système cible pour permettre l'observation d'événements lors de l'injection de fautes.

4.2.2.1 Charge de fautes

Nous avons vu que la technique d'injection de fautes retenue dans le chapitre 3 consiste à remplacer la valeur d'un paramètre par une valeur corrompue. Cette valeur est prédéfinie et

appartient à l'intervalle de définition du paramètre qu'elle remplace. Les valeurs corrompues sont alors substituées à la valeur initiale lors de l'exécution de la fonction du noyau appelée par le pilote. Afin de réduire la durée de nos campagnes d'expériences à quelques heures, nous avons focalisé la corruption des paramètres de fonction à un ensemble restreint de valeurs spécifiques correspondant aux valeurs les plus remarquables sur l'intervalle. Toutefois, cet ensemble pourra être enrichi pour l'obtention de résultats supplémentaires. En effet, la prise en compte d'un nombre plus important d'éléments corrompus révèle des comportements supplémentaires intéressants pour la sûreté de fonctionnement (voir chapitre 3).

Nous avons vu dans le Tableau 4-2 les classes de type de paramètres utilisés lors d'un appel aux fonctions du noyau *Linux*. Le Tableau 4-3 montre les bornes de leurs intervalles de validité. La valeur qui est substituée à la valeur originale du paramètre corrompu en dépend. Le Tableau 4-4 précise les valeurs corrompues injectées selon la classe considérée conformément à la méthode présentée dans le chapitre 3.

Type	Valeur corrompue 1	Valeur corrompue 2	Valeur corrompue 3
Pointeur lecture/écriture	*NULL*	*random( )*	*All bits = 1 (0xFFFFFFFF)*
Pointeur vers structure	*NULL*	*random( )*	*All bits = 1 (0xFFFFFFFF)*
Pointeur temporel	*NULL*	*random( )*	*All bits = 1 (0xFFFFFFFF)*
Entier signé	*INT_MIN(0x80000000)*	*0*	*INT_MAX(0x7FFFFFFF)*
Entier non signé	*0*	*INT_MIN(0x80000000)*	*ULONG_MAX(0xFFFFFFFF)*
court non signé	*0*	*SHRT_MIN (0x8000)*	*USHRT_MAX(0xFFFF)*
Identificateur de processus	*INT_MIN(0x80000000)*	*0*	*INT_MAX(0x7FFFFFFF)*
Descripteur de fichiers	*INT_MIN(0x80000000)*	*0*	*INT_MAX(0x7FFFFFFF)*
Mode	*INT_MIN(0x80000000)*	*0*	*INT_MAX(0x7FFFFFFF)*

Tableau 4-4 Valeurs corrompues par type de paramètre

Pour les trois premiers types, les valeurs limites et médianes de l'intervalle de validité sont considérées. Pour des pointeurs, l'ensemble de valeurs corrompues est : NULL, une valeur limite maximale et une valeur aléatoire dans l'espace des valeurs valides. Ces valeurs sont pertinentes pour un type de paramètres donné afin d'évaluer la robustesse d'une fonction. Typiquement, lors d'une étape de validation d'un logiciel, on teste une fonction prenant en paramètre un pointeur en remplaçant son paramètre par la valeur *NULL* et une valeur fixe dépassant l'espace d'adressage autorisé.

4.2.2.2 Charge de travail

Pour simuler une activité du système, nous considérons une charge de travail synthétique et modulaire combinant plusieurs processus d'activation, chacun stimulant un ou plusieurs des

pilotes installés. Comme spécifié dans la méthode présentée au chapitre 3, chaque processus effectue un ensemble d'opérations élémentaires sur un périphérique spécifique.

Par illustrer ces propos, le *Tableau 4-5* décrit les charges de travail de deux pilotes de périphériques : Un pilote d'une carte réseau de la firme *SMC* et le pilote générique aux lecteurs de disque compact de la version 2.4.18 de *Linux*.

Opérations	Description	SMC-ultra	CD-ROM
Désinstallation	Processus supprimant le pilote du système	unregister_netdev, release_region	unregister_sysctl_table, kfree
Installation	Processus inserant le pilote dans le système	register_netdev, request_region, NS8390_init	register_sysctl_table, kmalloc
requêtes	Effectue l'ouverture et l'utilisation des fonctionnalités du pilote	ei_interrupt ei_open ethdev_init free_irq ioport_resource isapnp_find_dev isapnp_present kfree printk request_irq	__generic_copy_from_user __generic_copy_to_user check_disk_change printk sprintf
Non mis en œuvre	Symboles n'ayant pu etre activé par la charge de travail	N/A	proc_doint proc_dostring

Tableau 4-5 Symboles activés par les deux charges de travail spécifiques

Les phases de désinstallation et d'installation permettent d'activer les symboles contenus dans la partie « politique » (voir chapitre 1) des deux pilotes présentés. Au plan logiciel, elles se composent d'instructions administrateur pour l'installation et la configuration des pilotes de périphériques. Ces instructions sont très proches d'un pilote à l'autre.

Les requêtes (au niveau de l'API) des charges de travail activant les périphériques de ces deux pilotes sont toutefois très différentes. En effet, pour effectuer des opérations sur le pilote de la carte réseau, il est nécessaire d'utiliser des commandes utilisant le réseau du banc d'essai. Pour le pilote de lecteur de disque compact, nous avons considéré des opérations sur un CD inséré dans le lecteur et sur le lecteur lui-même (montage et démontage). De cette manière, des fonctions de la DPI très différentes sont activées par la charge de travail.

Il est important de noter que l'ensemble de ces opérations ne suffit pas toujours à activer l'intégralité des fonctions utilisées dans un pilote. Par exemple, dans le Tableau 4-5, alors que dans le cas du pilote de la carte réseau nous avons activé l'ensemble des fonctions grâce à notre charge de travail, certains symboles du pilote de lecteur de disque compact n'ont jamais été utilisés comme les symboles *proc_doint* et *proc_dostring* (fonctions spécifiques de lecture pour les opération de contrôle du système -*sysctl*-).

Dans le cas du pilote pour le lecteur de disque compact, la charge de travail est limitée par les contraintes physiques du banc de test. Certaines opérations du pilote nécessitent la présence de composants matériels particuliers absents de notre architecture et les fonctions du noyau appelées dans ces sections « conditionnelles » ne peuvent donc pas être évaluées.

Comme indiqué au chapitre 3, une seconde exécution de charge de travail est effectuée sans injection de faute afin d'améliorer l'évaluation de l'impact de la faute injectée sur l'ensemble du système. L'objectif est d'améliorer le diagnostic dans le cas où aucune manifestation ne serait observée. Une expérience où la première exécution de la charge de travail n'a permis aucune observation est alors confortée par cette deuxième exécution. En effet, elle indique que le système répond correctement à des charges supplémentaires sans qu'un événement vienne troubler la réalisation complète de celui-ci. Dans les expériences rapportées dans ce chapitre, la charge de travail qui est exécutée pour améliorer le diagnostic est identique à la charge de travail employée pour les expériences d'injection de faute. Nous y ferons référence dans la suite de ce document par « seconde exécution de la charge de travail ».

4.2.3 Injection de fautes

Il est important de préciser que l'implémentation de cette partie de code de l'outil est dépendante du système d'exploitation et de l'architecture de la machine ciblée. Pour *Linux*, elle est concrétisée sous la forme d'un module qui agit au niveau de la pile d'exécution du processeur. La détermination de l'instant d'injection est effectuée grâce aux interruptions mises à disposition par le noyau.

Les deux principales difficultés de la mise en œuvre de l'outil résident dans l'interception des appels aux fonctions du noyau et la corruption de leurs paramètres. En effet, les travaux basés sur une injection de fautes externes au noyau, comme dans [Koopman & DeVale 1999], [Kalakech *et al*. 2004], [Jarboui *et al*. 2002a] et en partie [Rodriguez *et al*. 2002], effectuent l'injection de fautes au niveau de l'interface applicative ou des applications. L'interception des appels systèmes effectués au niveau de l'API par les applications est aujourd'hui maitrisé par des outils universitaires ou commerciaux. Cependant lors d'une interception d'une fonction au niveau de la DPI, le problème se complexifie et, aujourd'hui, seul *Nooks* (voir chapitre 1) le permet dans une certaine mesure.

L'outil que nous avons développé permet d'effectuer une injection de fautes à l'intérieur du système, entre le noyau et ses extensions, comme les pilotes de périphériques. Cette injection a été élaborée pour être la moins intrusive possible. Afin de mieux appréhender la technique d'injection de fautes utilisée, il est nécessaire de fournir quelques explications sur le fonctionnement du passage de paramètres à un symbole lors de l'exécution du code associé à un pilote de périphérique.

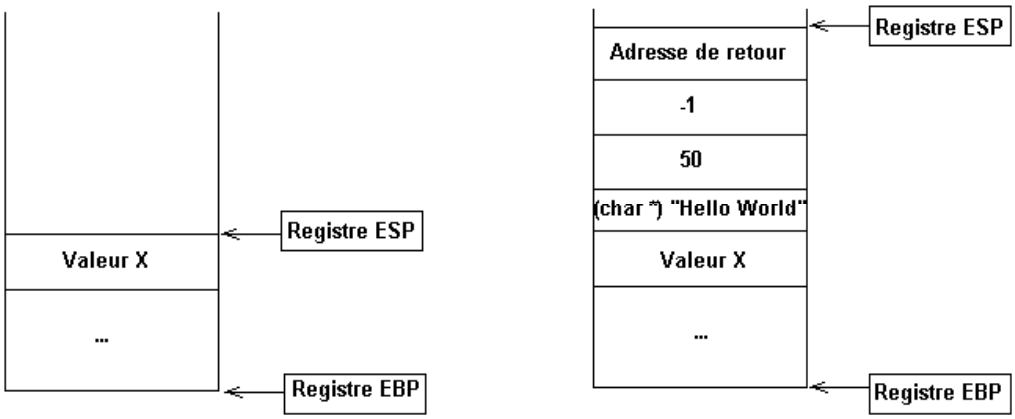
Le processeur emploie une pile résidant dans la mémoire RAM (*Random Access Memory*) de la plate-forme pour stocker diverses données, y compris les paramètres des appels aux fonctions de la DPI. Les instructions devant être exécutées sont stockées dans un autre secteur de la mémoire. La pile est accessible via plusieurs registres du processeur contenant, entre autres, les adresses du sommet (registre *ESP*, pour *Extended Stack Pointer*) et de la base de la pile courante (*EBP* pour *Extended Base Pointer*) représentés sur la Figure 4-2a. Ces registres peuvent être manipulés directement par des instructions proche du langage machine avec le langage d'assemblage (voir [Manuel Intel]).

Lorsqu'une fonction est appelée en langage C, le processeur enregistre les paramètres dans la pile, puis il exécute le code de la fonction. Lors de l'empilement des paramètres, il incrémente également le registre ESP pour que celui-ci pointe toujours sur le sommet de la pile. Ainsi, la

fonction appelée peut consulter la pile pour obtenir les valeurs de ses paramètres par l'intermédiaire du registre ESP. Les paramètres se trouvent à une profondeur qui dépend du nombre et de la taille des paramètres de la fonction.

L'exemple de la Figure 4-2 représente l'évolution de la pile lors de l'appel d'une fonction *foo* dont la signature est *foo(-1, 50, "Hello World")*. Le processeur empilera un à un les paramètres de *foo*, en incrémentant le registre ESP. Lors de l'appel à la fonction *foo*, l'adresse de retour (sur 4 octets) est empilée également pour indiquer la prochaine instruction à exécuter une fois la fonction terminée. Grâce à ce fonctionnement, nous pouvons obtenir le premier paramètre en consultant la pile à une profondeur de quatre sous le sommet de la pile. Le second paramètre est situé à une profondeur de quatre plus la taille en octet du premier paramètre. Le procédé se répète pour les autres paramètres de la fonction.

Dans le cas de *Linux*, la possibilité de placer des interruptions dans le système permet d'insérer un traitement lors de l'exécution du code noyau. Pour une expérience donnée, nous avons placé une interruption sur un des symboles.



a) Pile avant appel **b) Pile après appel**

Figure 4-2 Evolution du contenu de la pile lors d'un appel à un symbole Linux

La Figure 4-3 illustre ce principe. Quand le symbole de la DPI visé commence à être exécuté, une interruption est levée par le processeur lors du passage sur le point d'arrêt placé sur la première instruction correspondant à la fonction appelée par le pilote (*PUSHAD*). L'exécution courante est interrompue pour substituer la valeur d'un des paramètres de la pile par une valeur du modèle de fautes, puis l'exécution reprend avec cette nouvelle valeur.

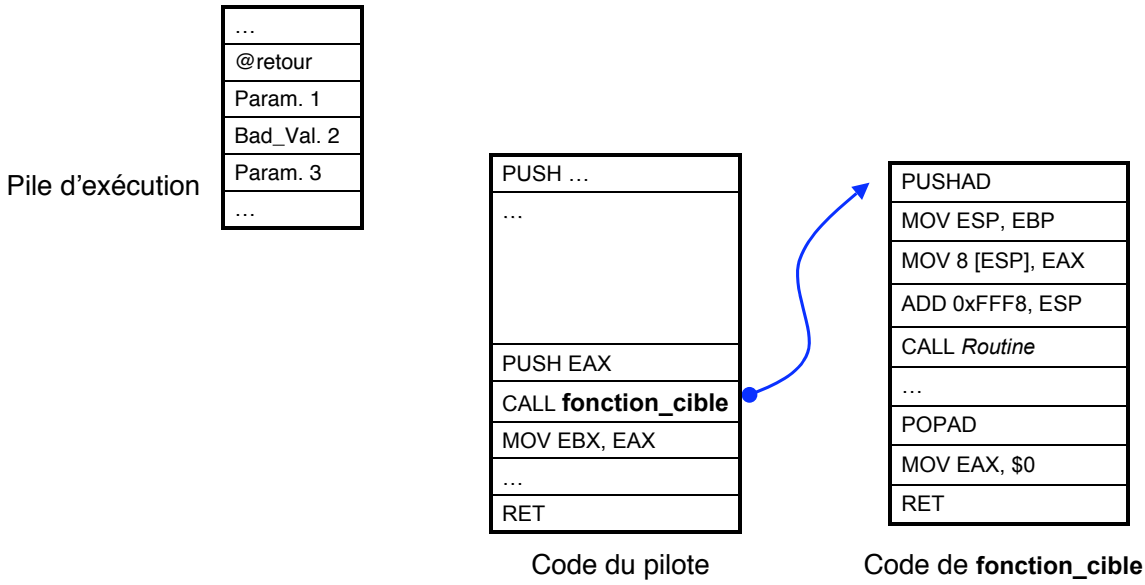


Figure 4-3 Interruption d'un appel à une fonction du noyau Linux

Quand la faute a été appliquée une fois, le module d'injection de pilote est inactivé, l'interruption placée sur le symbole est levée. La corruption provoquée correspond donc à une faute transitoire (par opposition à permanente). Son objectif est de représenter un unique mauvais appel à une fonction du noyau par un pilote défaillant, par exemple. Une fois la faute injectée, l'outil d'injection est désactivé. Afin d'identifier les symboles employés par les pilotes du système, nous avons développé des procédures qui extraient automatiquement leurs noms à partir du fichier contenant le code objet du pilote. Puis, grâce à une liste référençant tous les symboles *Linux* et leurs signatures, nous pouvons déterminer quelle injection doit être effectuée sur ces symboles. Cette liste est établie grâce à l'étude préalable décrite dans le paragraphe 4.1.

Pour effectuer aisément les opérations décrites précédemment, nous avons opté pour une solution logicielle qui se présente sous la forme d'un module logiciel *Linux*. Son rôle consiste à placer le point d'arrêt en mémoire sur la première instruction concernant le code du symbole ciblé puis à introduire l'injection par traitement de la pile. Cette solution apporte un faible niveau d'intrusion lors de la surveillance (interruption « basique » du noyau), lors du traitement (injecteur en mode noyau) et de l'injection (instructions d'assemblage sur la pile).

Pour mettre en œuvre une faute permanente, il suffirait de réactiver le point d'arrêt sur le symbole visé pour réinjecter la faute à chaque utilisation de celui-ci. Le module d'injection de faute peut aisément implémenter ce type de faute. Toutefois, la mise en place d'une faute permanente dans le noyau et l'activation répétée de la phase d'injection de faute augmente l'intrusivité de l'outil.

4.2.4 Observations et mesures

Nous avons vu au paragraphe 3.3.3 qu'il existe plusieurs niveaux d'observation d'un système :

- À l'intérieur du noyau, principalement grâce au code retour de la fonction ciblée du noyau,
- À l'extérieur du noyau mais sur la machine cible, comme les mesures effectuées sur la charge de travail ou le comportement du système.
- Sur une machine différente de la machine ciblée, observant le comportement global de la machine.

Nous détaillerons ces différentes observations et les mesures effectuées dans ce paragraphe.

Observati` ns internes

Les codes retour des fonctions constituent une information cruciale dans l'évaluation de la robustesse d'un symbole. Ils indiquent la terminaison correcte ou incorrecte de la fonction. Lors d'une terminaison incorrecte, la cause présumée peut être indiquée dans le code retour d'erreur. Si la fonction détecte la valeur corrompue, elle doit retourner un code d'erreur. Dans le cas contraire, elle ne retourne pas de code d'erreur et « ignore » la présence de la faute.

Pour obtenir ces observations internes, la technique que nous avons conçue est identique à celle utilisée lors de l'injection de faute. Le code retourné par le symbole du noyau soumis à une faute est observé sur la pile lors de la terminaison de la fonction. En effet, une fois l'injection réalisée nous plaçons un nouveau point d'arrêt sur le dernier élément du code correspondant au symbole ciblé. Les codes retour négatifs ont une signification particulière dans le noyau Linux. Par exemple, la valeur *–22* correspond à l'erreur *invalid value*. Ces codes d'erreur sont définies dans le fichier *errno.h* du code source de *Linux*. Cette information permet d'améliorer la compréhension du diagnostic du système vis-à-vis de la faute injectée.

La méthode utilisée par le compilateur GCC pour renvoyer le code retour consiste à le placer dans un des registres entiers à usage général nommé EAX. Cette valeur est toujours présente même si la fonction appelante ne l'utilise pas. Ainsi, au retour de la fonction, le code d'erreur se trouve toujours dans le registre EAX et nous pouvons extraire sa valeur à la fin de l'exécution de la fonction.

Néanmoins ces codes ne sont pas les seuls événements observables et les fautes injectées entraînent d'autres manifestations ou messages d'erreur provenant du système, comme la levée d'une exception ou l'entrée en mode panique. Ces messages d'erreur sont rassemblés à la fin de chaque expérience. Les exceptions et les modes paniques sont relevés en consultant le journal (*log*) de la machine cible.

Observati` ns externes

Au sein de la plate-forme expérimentale actuellement mise en œuvre, les observations externes concernent exclusivement le comportement des deux charges de travail composées d'un ensemble de requêtes applicatives. Ces requêtes sont liées aux symboles ciblés par la campagne. Cependant, le contrôle du bon déroulement des autres requêtes, celles qui ne sont pas destinées aux symboles visés par l'expérience en cours, constituent une bonne mesure du comportement du système d'exploitation vis-à-vis de son environnement applicatif.

Pour observer ce comportement, les résultats des requêtes effectuées au niveau applicatif sont relevés par le module d'observation. Chacune de ces requêtes retourne un code retour pour notifier leur bonne terminaison ou retourne un code d'erreur correspondant à l'origine de la non terminaison de leur tâche. De plus, pour certaines opérations, un contrôle est effectué par rapport au résultat attendu de la charge de travail.

Par exemple, certaines requêtes sont constituées d'opérations d'entrée/sorties sur le disque dur. Les résultats sont récupérés via l'opération de lecture et d'écriture. Le module d'observation compare l'état du disque (ou plutôt de la zone du disque) par rapport au résultat attendu. Cet « oracle » est déterminé lors de la conception des charges de travail. C'est un résultat reproductible de l'exécution de la charge de travail sur la machine évaluée dans un environnement sans injection de fautes.

Au chapitre trois, nous avons défini trois options pour la terminaison de la charge de travail. Elle est peut être non terminée, terminée correctement ou terminée avec des erreurs. Si une des requêtes effectuées dans la charge de travail n'a pas été effectuée correctement nous considérons une terminaison erronée de la charge de travail (terminaison de charge incorrecte). La seconde exécution de la charge de travail (re-exécution de la charge de travail sans faute) permet, pour chaque expérience, d'évaluer les capacités du système à recouvrer, ou non, un état stable de façon autonome. Les résultats de cette deuxième exécution sont évalués de manière identique à ceux de la première (non terminaison, terminaison et valeurs fournies incorrectes ou terminaison et valeurs fournies correctes). La latence de manifestation de l'effet de la faute peut entraîner l'apparition d'observations uniquement lors de l'exécution de cette seconde charge.

Comme nous l'avons présenté dans le chapitre 3.3.3, le journal de la machine cible ne permet pas d'observer certains événements. Alors, la machine observatrice relève ces événements, pour la plupart des blocages noyaux, grâce à l'observation du déroulement des expériences sur la machine ciblée. Si le temps d'exécution de l'expérience dépasse la fenêtre temporelle d'observation (fixée par retour d'expérience), la machine ciblée est considérée bloquée.

Gesti` nnaire de redémarrage matériel

La nature même des expériences peut provoquer des erreurs ayant un fort impact sur le déroulement d'une campagne d'injection de fautes. Ainsi, les erreurs provoquées peuvent entraîner des dysfonctionnements importants du noyau, interrompant le déroulement des exécutions. Pour pallier ce problème, nous avons mis en place un dispositif matériel affecté à un port de la machine observatrice gérant le redémarrage matériel des machines appartenant au banc de test. Ce dispositif est relié physiquement, par un câble électrique, aux branches de restauration (*reset*) des machines cibles. Si la machine observatrice relève un blocage de la machine sur lequel l'expérience est effectuée, le gestionnaire de redémarrage matériel envoie une impulsion électrique sur la broche de redémarrage de la machine bloquée, provoquant ainsi sa réinitialisation de façon automatique.

Pour relever un blocage noyau, au début de chaque expérience, la machine cible envoie un signal à la machine observatrice qui initialise une temporisation (voir figure 3-6). À la fin de chaque expérience, la machine cible est redémarrée, ce qui ré-initialise la temporisation, signalant que le système n'est pas bloqué. Si le système se bloque ou s'il ne peut plus redémarrer de manière logicielle, il ne peut pas réinitialiser le temporisateur. Si celui-ci expire,

la machine observatrice provoque un redémarrage matériel de la machine cible par une impulsion électrique.

Mesures temp` relles

Ces mesures concernent actuellement trois temps d'exécution :

- Le temps de redémarrage de la machine après une expérience,
- Le temps d'exécution de la charge de travail,
- Le temps d'exécution de la deuxième charge de travail.

Pour les deux dernières, les mesures sont effectuées par la machine cible et sont insérées dans son journal. La machine relève le démarrage et la fin de chaque exécution de la charge de travail.

Pour le temps de redémarrage, deux cas sont distingués. Si la machine a redémarré sans l'intervention du gestionnaire de redémarrage matériel, les mesures sont effectuées par la machine cible comme pour les charges de travail. Si la machine ciblée n'a pu redémarrer de manière autonome, le temps de redémarrage est calculé à partir du moment où l'impulsion électrique est envoyée à la machine bloquée pour la faire redémarrer.

4.2.5 Application à *Windows XP*

La mise en œuvre expérimentale sur un système différent (système d'exploitation et plateforme matérielle) implique des modifications dans certains modules de l'outil d'injection de fautes. En effet, la principale difficulté d'une technique d'évaluation de la robustesse d'une interface par corruption de paramètres consiste à intercepter l'appel que l'on souhaite corrompre.

Sous *Windows*, un échange, via une IRP, entre le noyau et un pilote peut être intercepté par des pilotes « filtres hauts ». Le rôle de ces pilotes filtres est justement le prétraitement d'une fonction du noyau lors de son appel par le pilote filtré et le post-traitement de son code retour. La corruption peut donc être effectué à ce niveau, comme les observations internes. Une fois l'interception réalisée grâce aux filtres, la corruption est appliquée de manière similaire à l'implémentation sous *Linux*. Les observations externes sont effectuées au niveau applicatif dans un journal d'expérience modifié au niveau applicatif.

Une autre technique d'interception des interactions entre un pilote et le noyau du système d'exploitation consiste à modifier la table des fonctions externes du code binaire d'un pilote. Cependant, cette technique risque de se relever plus intrusive que celle présentée ici. Le temps d'exécution des fonctions « de déroutage » modifie de manière plus importante le comportement du système qu'une simple routine en Assembleur x86, ne serait-ce que par le temps d'exécution de l'injection.

Sur la plupart des systèmes de la famille *Windows*, la possibilité de modifier la base de registre du système d'exploitation permet d'intercepter une partie des interactions entre le noyau et le pilote. Cela peut constituer une approche intéressante au niveau de la faible intrusivité de la technique. Cependant, la technique nécessite le plus souvent de développer un pilote spécifique pour corrompre le pilote sélectionné pour être le pilote défaillant.

Les types du système d'exploitation *Windows* étant sensiblement identiques à ceux de *Linux*, le modèle de fautes ne doit être que légèrement modifié pour prendre en compte ces nouveaux éléments.

4.3 Dér` ulement d'une campagne d'expériences

Nous allons décrire dans ce paragraphe le déroulement étape par étape d'une campagne d'expériences d'injection de fautes. Après avoir détaillé l'étape préliminaire consistant à déterminer les fonctions ciblées par la campagne, nous décrivons ensuite le déroulement et l'exécution des expériences d'injection de fautes. Enfin, nous précisons comment le stockage des données est effectué à la fin de chaque expérience.

4.3.1 Détermination des symboles ciblés

La première étape pour le lancement d'une campagne d'injection de fautes permettant l'évaluation de la robustesse d'une interface est de définir quelles parties de l'interface seront visées par la campagne. Il existe trois points de vue pour aborder une interface. Ils sont illustrés sur la Figure 4-4. La partie de gauche représente les symboles utilisés par les pilotes présents dans le système. La partie de droite représente les symboles présents dans la DPI du noyau, classés par service.

Le premier point de vue consiste à considérer l'ensemble de l'interface pilote noyau du système (*Service A* + *Service B* + *Service C* + ...). Les symboles utilisés par le système global seront donc tous évalués lors de la campagne. Ce point de vue est théoriquement le plus intéressant car il présente l'avantage de cibler l'intégralité de l'interface que l'on souhaite évaluer. Toutefois il se révèle peu réaliste en pratique. En effet, un grand nombre de symboles

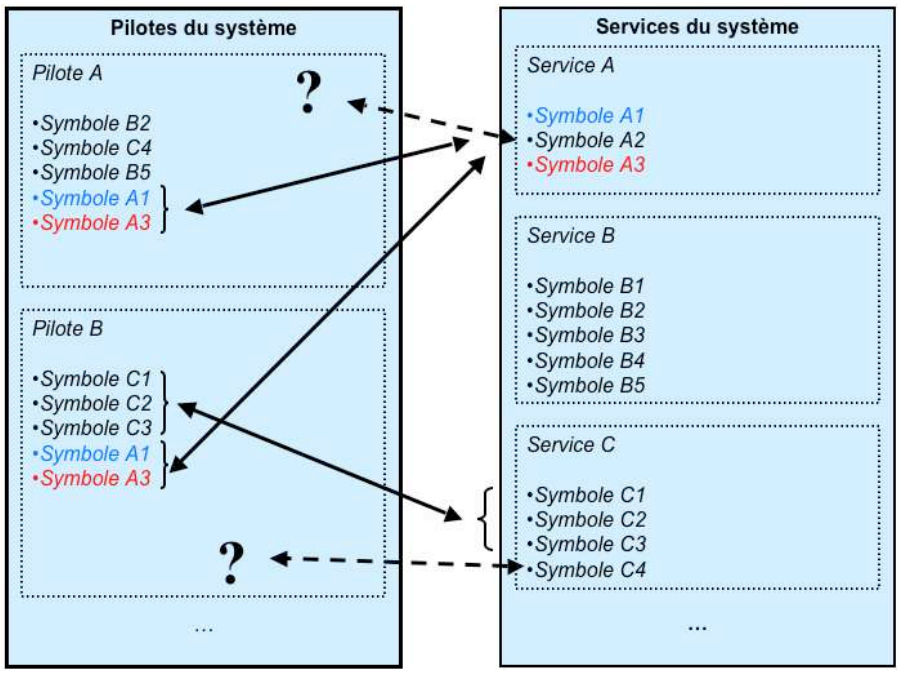


Figure 4-4 Répartition des symboles dans les pilotes

de l'interface ne sont pas utilisés par la machine cible. D'une part, car on ne peut pas installer tout le matériel possible et gérer par le noyau sur une même machine. D'autre part, car la charge de travail devrait être d'une complexité équivalente aux nombres de symboles ciblés. Ces cas sont illustrés par les symboles *A2* et *C4* signalés sur la figure par un point d'interrogation.

Le deuxième point de vue est une version allégée du premier. Plutôt que de considérer l'ensemble de l'interface, on peut considérer à part chaque service de l'interface (*Service A* ou *Service B* ou *Service C* ou ...). Les fonctions du noyau fournies aux programmes privilégiés sont regroupées en services distincts (Le service *A* inclut les symboles *A1*, *A2* et *A3*). L'analyse peut aussi être réalisée sur la base de ce partitionnement. Les services peuvent être comparés et lors d'un changement de noyau (par exemple de *Linux 2.2.20* à *2.4.18*), l'évolution de l'interface peut être localisée sur un service précis. Toutefois, l'inconvénient d'une telle approche reste la difficulté d'activer toutes les fonctions d'un service donné. La difficulté reste cependant plus abordable que celle du premier point de vue, ciblant l'intégralité de la DPI. En effet, les pilotes du système n'utilisent pas toutes les fonctions du noyau du système. Mais si l'on utilise des architectures identiques pour comparer différents systèmes d'exploitation ou différentes versions d'un même système d'exploitation, on peut considérer que la couverture de l'interface est proche : les mêmes fonctionnalités de la DPI sont évaluées. On peut alors comparer les résultats provenant de différents noyaux ou différentes versions d'un même noyau.

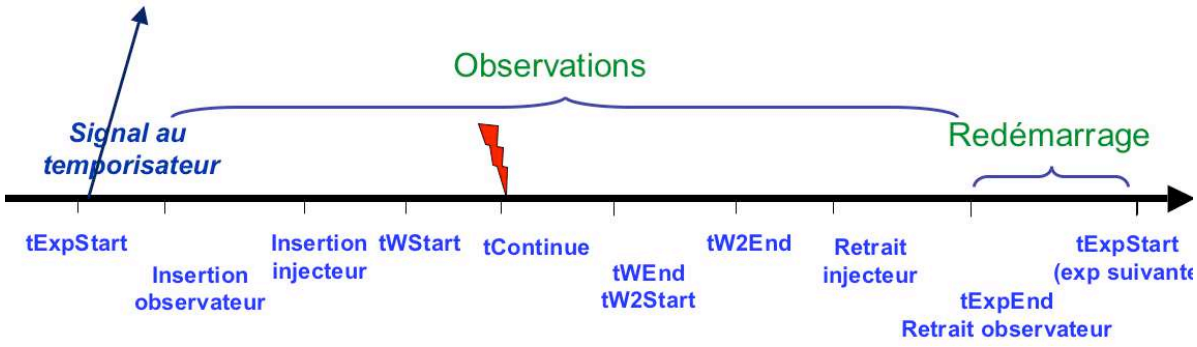
Le troisième point de vue consiste à étudier les symboles à partir des pilotes installés sur le système (*Pilote A* + *Pilote B* + ...). En effet, les pilotes de périphériques du système utilise un ensemble de fonctions données. La figure illustre la complétude de cette technique : tous les symboles appelés dans le code des pilotes installés du système ont une probabilité élevée d'être utilisés. Ces fonctions seront pour la plupart activées lors de nos expériences si la charge de travail est suffisamment explicite. Les pilotes utilisent des fonctions appartenant à plusieurs services différents. On peut alors comparer la robustesse des fonctions de services similaires de l'interface entre deux versions d'un noyau. De plus, nous pouvons comparer les résultats de pilotes ayant une implémentation différente mais effectuant des opérations similaires (par exemple, comparer deux pilotes d'une même carte réseau ou deux pilotes de cartes différentes).

Pour extraire les symboles utilisés par un pilote à partir de son fichier objet, nous utilisons principalement la commande *nm* de *Linux*. Cette commande met en évidence les fonctions non résolues du fichier objet. Les fonctions appartenant au noyau sont notées de manière spécifique dans un tableau et la liste des symboles utilisés par un pilote peut alors être établie.

Pour extraire les symboles d'un service, nous nous référons à l'étude préalable du système (voir paragraphe 3.3.1). En effet, lors de l'étude des signatures des fonctions, les fonctions sont classées dans un service du noyau. Chacune de ses fonctions est implémentée dans un des nombreux fichiers composant le code du noyau *Linux*.
La signature des symboles, principale information concernant la fonction est obtenue à partie du fichier référence de l'interface. Les informations dépendent de la documentation du noyau sur l'interface de programmation entre les pilotes et le noyau.

4.3.2 Conduite de la campagne d'injection de fautes

La première expérience peut être lancée lorsque les symboles ciblés sont déterminés et la charge de travail prête à les activés. Grâce au gestionnaire de programmes lancés au démarrage de *Linux* présents dans */etc/rc2.d/*, nos expériences sont lancées à chaque démarrage de la machine. La Figure 4-5 présente le chronogramme d'une expérience classique d'injection de fautes.



Étape	Événements	Implémentation *Linux*
tExpStart **Insertion X**	Vérification du système. Initialisation du compte à rebours. Mise en place de l'outil et choix de la faute à injecter.	Lancement de l'utilitaire *e2fsck* et vérification de l'intégrité du système. Placement du point arrêt.
tWStart	Démarrage de la charge de travail.	
DPI call	Interception de l'appel à la fonction du noyau ciblée et injection de la faute. Observation du code retour d'erreur de la fonction.	Levée d'une interruption et injection de la faute.Placement du point d'arrêt pour l'observation du code retour.
tContinue	Reprise de la charge de travail après exécution de la fonction en présence de fautes.	Observation au niveau interne du code retour d'erreur.
tResponse	Observation des événements perceptibles au niveau externe.	Relevé des résultats fournis par la charge de travail.
TWEnd	Terminaison de la charge de travail.	Relevé de la terminaison de la charge
tW2Start	Lancement de la seconde charge de travail.	
tW2End	Terminaison de la seconde charge de travail.	Relevé de la terminaison de la charge et des résultats fournis.
TExpEnd **Retrait X**	Fin de l'expérience courante et redémarrage.	Suppression des modules de l'outil.

Figure 4-5 Chronogramme d'une expérience

Le cas illustré dans la figure concerne une expérience où le système ne s'est pas bloqué et la charge de travail s'est terminée normalement. Nous allons voir étape par étape le déroulement d'une expérience d'injection de fautes. Chaque expérience dure entre 2,5 et 5 minutes, le pire cas correspondant aux expériences ayant amené un blocage noyau. Une expérience débute par le lancement d'utilitaires comme *e2fsck* vérifiant l'intégrité du disque dur. En effet, les redémarrages à répétition brutaux peuvent engendrer des erreurs dans le système de fichiers et provoquer un état instable permanent de la machine. Le temporisateur *Timer Signal* (n) est initialisé pour signaler un redémarrage correct de la machine pour l'expérience courante.

La gestion du compte à rebours et son traitement se fait grâce à un script écrit dans le langage interprétatif *Expect* et exécuté sur la machine observatrice. Le compte à rebours est lancé lors de la première expérience. En pratique, sa valeur vaut quatre fois le temps d'exécution de la charge de travail utilisée afin de conserver une marge de « sécurité » significative. En effet, cette marge permet de maintenir un intervalle d'observation suffisamment grand pour observer toutes les réactions du système en présence de la faute injectée.

Les outils d'observation et d'injection sont chargés en deux étapes :

- D'abord le module d'observation (*Loading observer*),
- Puis celui d'injection de fautes (*Loading injector*).

Ce dernier reçoit comme arguments les paramètres de l'expérience avec la signature du symbole ciblé, le paramètre visé et l'indice de la valeur invalide à injecter. Le rôle du module injecteur sous *Linux* consiste à placer, dans un premier temps, un point d'arrêt (*breakpoint*) en mémoire sur la première instruction concernant le code du symbole ciblé.

La charge de travail est lancée à *tWStart*. Lors du passage sur la fonction visée par l'expérience courante, elle va provoquer la levée d'une interruption après le premier passage sur le point d'arrêt placé sur le code associé au symbole ciblé. L'injection de la faute puis la mise en place du deuxième point d'arrêt pour l'observation de la valeur du code retour de la fonction est effectuée à cet instant-là. La fonction termine alors son traitement avec une valeur de paramètre corrompue. Une deuxième interruption est levée pour permettre le relevé du code retour, puis l'exécution de la charge de travail reprend (*tWContinu*).

Lorsque la première charge de travail se termine (*tWEnd*), la charge témoin est lancée (*tW2Start*). Elle se termine à son tour (*tW2End*) puis les modules d'injection et d'observation sont déchargés. Les fichiers de résultats sont stockés dans un journal d'expérience pour un futur archivage dans une base de données. Enfin, la machine est redémarrée pour le lancement de l'expérience suivante.

4.3.3 Stockage des données

La diversité des observations effectuées et l'instabilité de la machine cible compte tenu des fautes injectées, nous ont amené à mettre en place un système de stockage de données externes aux machines cibles.

À chaque terminaison d'expérience, les relevés des observations internes et externes effectués par la machine cible sont récupérés par la machine observatrice qui les rajoute à un journal d'expérience contenant ses propres observations externes. Toutefois, dans le cas de fautes

entraînant une instabilité de la machine cible, les relevés de la machine cible peuvent être indisponibles. En effet, certaines fautes peuvent affecter l'intégrité du système de fichiers. C'est notamment pour pallier ce problème que les données sont régulièrement transférées sur la machine observatrice. À la fin de chaque campagne, les données de toutes les expériences sont stockées dans une base de données en vue d'analyses ultérieures.

4.4 Résultats et Analyses

Les observations recueillies offrent une base pour divers types d'analyses. En effet, une même série d'observations pourra être interprétée de manière différente en fonction de son contexte. La criticité de l'occurrence d'un événement imprévu est différente entre un système critique et celui d'un ordinateur personnel. Nous verrons que les analyses peuvent être effectuées en fonction de l'impact sur la sûreté de fonctionnement que l'on accorde aux combinaisons de comportements observés.

Dans la pratique, les différentes interprétations des mesures dépendent du contexte spécifique où le noyau doit être intégré. Par exemple, quand on favorise le comportement de la charge de travail, les notifications d'erreurs provenant du système d'exploitation peuvent être des comportements appropriés si la charge n'est pas affectée. Inversement, les arrêts ou la corruption de la charge applicative est acceptable lorsque l'on recherche la disponibilité du noyau. Ce phénomène est accentué par l'observation de plusieurs événements durant la même expérience comme nous l'avons vu dans le paragraphe 3.4. Afin d'exprimer divers points de vue de sûreté de fonctionnement, l'analyse des résultats doit être menée de façon minutieuse et avec discernement. Nous illustrons dans ce paragraphe que les façons d'interpréter les observations modifient les mesures de sûreté de fonctionnement.

Nous présentons et analysons un ensemble de résultats obtenus avec le banc d'essai pour trois pilotes représentatifs fonctionnant avec le noyau *Linux*. La présentation des résultats est limitée à un ensemble choisi afin de faciliter l'exposition des analyses. Nous soulignons volontairement les résultats de deux pilotes soutenant une carte réseau différente. Les pilotes liés au réseau représentent une grande partie du code d'un système d'exploitation [Godfrey & Tu 2000] et sont souvent à l'origine de dysfonctionnement du système. Nous considérerons également un autre pilote, gérant une carte son, pour compléter les résultats et augmenter la couverture des symboles de la DPI évalués. Dans le même objectif, un ensemble de services les plus courants du noyau utilisés par la plate-forme matérielle de la machine ciblé a également été évalué.

L'objectif des campagnes d'expériences menées est d'évaluer le comportement de :

- Deux pilotes de périphériques différents fonctionnant sur la même version du noyau. Il s'agit du pilote pour la carte son *Sound Blaster* (*SB*) et le pilote pour la carte réseau *SMC* (*SMC*) sur le noyau *Linux 2.2.20*.
- Deux réalisations de la même fonctionnalité fonctionnant sur le même noyau. Cela concerne les pilotes pour les cartes réseau *SMC* et *Ne2000* (*NE*) sur le noyau *2.4.18*.
- Le même pilote fonctionnant sur deux versions différentes du noyau. Il s'agit des deux versions du pilote *SMC*.

Les campagnes d'expériences ont permis l'évaluation de près de 150 symboles, ce qui représente environ 30% de l'ensemble des fonctions disponibles du noyau, tous services confondus. Cependant, il est important de noter qu'il s'agit des fonctions les plus fréquemment utilisés du système d'exploitation *Linux.*

Nous verrons dans ce paragraphe que l'analyse préliminaire des résultats observés au cours des campagnes menées met en évidence les limites de l'observation mais permet toutefois une première analyse des résultats fournis par l'outil. L'application des stratégies d'analyses présentées au paragraphe 3.4 va permettre d'améliorer notre interprétation du comportement du système en présence de fautes. Enfin, nous verrons comment l'analyse détaillée permet d'analyser de manière précise ce comportement dans l'objectif de comprendre l'origine de la propagation de fautes provenant des pilotes au reste du système.

4.4.1 Analyse préliminaire

Le Tableau 4-6 illustre la distribution des résultats obtenus en considérant la stratégie du premier événement lorsque plusieurs événements sont relevés au cours de la même expérience. En plus des événements spécifiques précédemment définis (voir *Tableau 3-3*), la table inclut également deux types de caractéristiques intéressantes :

- Non Activée (*Not Act.*): Les fautes injectées n'ont pas pu être activées. La charge de travail n'a pas activé la fonction sur laquelle la faute devait être injectée.
- Aucune Observation (*No Obs.*): Aucune notification d'événements et aucun mode de défaillance n'ont été observés alors que la faute a été activée.

Naturellement, quand la faute n'est pas activée, aucun événement particulier lié à la faute injectée ne peut être observé.

4.4.1.1 Les limites de l'observation

La proportion des expériences où les fautes n'ont pas été activées change de manière significative entre les pilotes et les versions de *Linux;* de 0% pour le pilote *SB 2.2* à 17% pour le pilote *SMC* (Voir Tableau 4-6).

Driver	Not Act.	No Obs.	EC	XC	KH	WA	WI
SB 2.2	0%	18%	47%	22%	3%	1%	9%
SMC 2.2	7%	22%	19%	23%	21%	0%	9%
SMC 2.4	17%	17%	21%	34%	11%	0%	0%
NE 2.4	14%	10%	15%	30%	17%	0%	13%

Tableau 4-6 Distribution des observations en considérant le premier élément relevé

Lorsque cette proportion est non nulle, cela peut signifier que les charges de travail respectives doivent être améliorées du point de vue de la testabilité afin de mieux contrôler les appels aux fonctions du noyau et en effectuer davantage. Cependant, ces taux sont très inférieurs à ceux qui sont rapportés dans d'autres études relatives au noyau *Linux,* comme dans [Gu *et al.* 2003].

Cette différence est due principalement à deux éléments. D'une part, dans notre cas, les fautes visent directement les paramètres des appels aux fonctions du noyau effectués par le pilote, au lieu du code exécutable correspondant au noyau. Toutes les parties d'un code exécutable ne sont pas utilisées par le processeur (par exemple à cause des structures répétitives et conditionnelles). D'autre part, les charges applicatives dédiées permettent une meilleure commandabilité des expériences en activant le pilote cible par les symboles que nous visons durant la campagne d'expériences.

L'interprétation des expériences sans observation (*No Obs.)* dépend étroitement du contexte spécifique où l'analyse est conduite. En effet, ces résultats peuvent être comptabilisés aussi bien comme étant positifs que négatifs, en fonction des points de vue correspondant à la notification d'événements, à la sûreté de la charge de travail ou encore à la disponibilité du noyau. Cependant, comme souvent dans toutes les approches expérimentales de ce type, des incertitudes demeurent toujours au sujet des véritables origines de ces résultats sans observation. En effet, l'outil d'injection de fautes peut être à l'origine de ce défaut de résultats, par un manque d'observation par exemple, ou une faute étrangère à notre système ayant affecté le système. Par conséquent, nous avons préféré adopter une approche conservatrice consistant à mettre à part ces résultats en vue d'analyses approfondies.

La deuxième exécution de la charge de travail a été mise en place pour réduire le nombre d'expériences sans observation et pour augmenter la confiance dans les interprétations effectuées. Une expérience privée de réaction du système indique dans certains cas une déficience de l'observation sur les expériences effectuées. Toutefois, il est fréquent que de tels résultats soient dus à des problèmes connexes difficiles à identifier :

- Le noyau n'a pas utilisé le paramètre corrompu,
- Le paramètre corrompu n'a aucun impact sur le noyau,
- L'erreur provoquée est masquée (dans notre cas, le plus fréquent est l'injection d'une valeur 0 sur un paramètre déjà égal à 0).

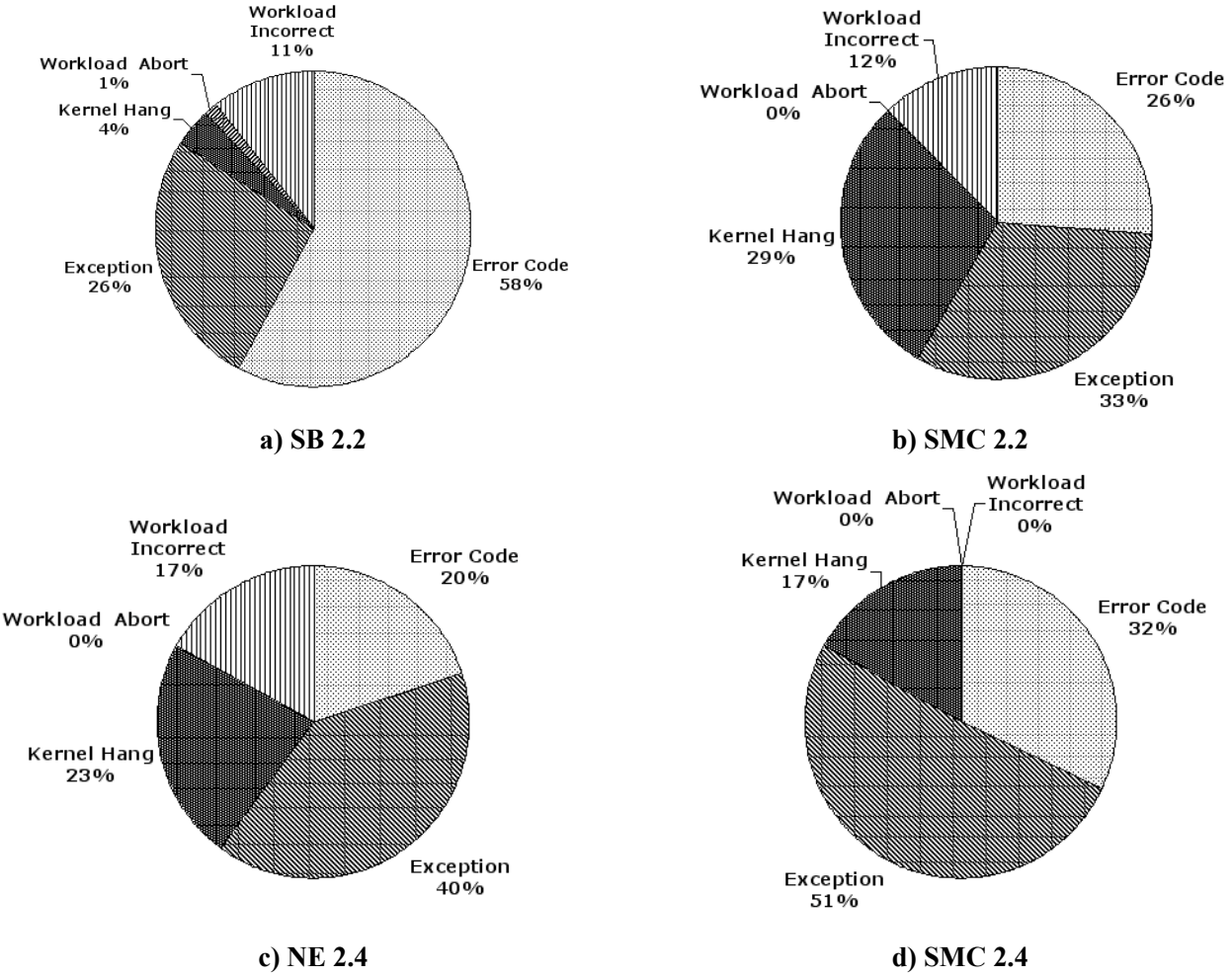
Cependant, bien que le nombre d'expériences où aucune observation n'est effectuée est plus important que celui où la faute n'a pas été activée, les valeurs de ces deux catégories sont sensiblement inférieures à celles qui ont été présentées récemment dans [Gu *et al.* 2003].

Une autre cause possible de pénurie d'observation peut être une implémentation de mauvaise qualité de la fonction. Par exemple, dans certains symboles du noyau, si la valeur d'un paramètre testé est considérée comme invalide, la fonction interrompt son exécution sans renvoyer de code retour d'erreur. Ce type d'implémentation répond mal aux exigences de sûreté de fonctionnement, en particulier pour établir des diagnostics, car elle peut conduire à des erreurs dont l'origine est difficile à déterminer.

Nous avons vu plus haut dans ce paragraphe que l'erreur peut être masquée car la valeur d'origine est identique à la valeur corrompue. Ces cas de masquage peuvent être également dû à des paramètres de type « drapeaux », où seuls certains bits particuliers du paramètre sont testés par la fonction, les autres étant ignorés.

4.4.1.2 Premières analyses

La Figure 4-6 illustre la distribution relative des événements observés en utilisant le critère du premier événement relevé. Un examen rapide de ces résultats montre une proportion très basse d'interruption de la charge de travail pour toutes les expériences effectuées. En effet, l'impact des fautes injectées dans les pilotes est rarement observé au niveau de la terminaison d'une application. Le plus souvent le système aura déjà relevé une anomalie avant la propagation à la charge de travail conduisant à une terminaison brutale de celle-ci. C'est pourquoi les résultats indiquent qu'un grand pourcentage des fautes est détecté en premier par le noyau (ceci inclut les événements de code retour et les levées d'exception). Dans la pratique, il est intéressant de différencier les notifications correspondant aux exceptions et code retour d'erreur car ce dernier est préférable au premier. En effet, un code retour d'erreur indiquant un mauvais usage de la fonction est nécessaire pour effectuer une action de rétablissement réussie du système.



a) SB 2.2 **b) SMC 2.2**

c) NE 2.4 **d) SMC 2.4**

Figure 4-6 Distribution des premiers événements observés

En conséquence, il est possible d'affirmer que les résultats pour le pilote *SB 2.2* sont meilleurs que ceux qui sont observés pour les expériences concernant les autres pilotes de carte réseau.

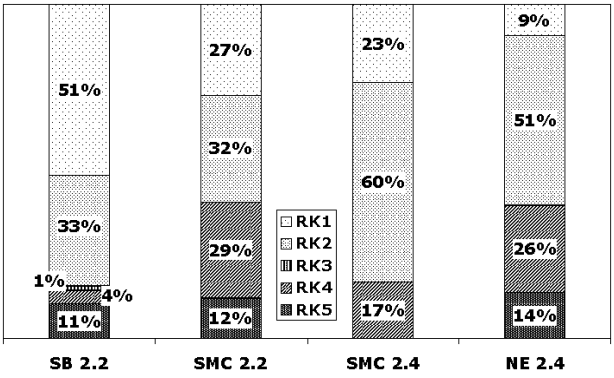
La comparaison des résultats obtenus par les pilotes *SMC* des deux versions du noyau indique clairement une amélioration de la robustesse en ce qui concerne la version *2.4*. Elle est

marquée par l'augmentation de la couverture des exceptions lors de l'injection de faute dans les symboles du pilote. Une conséquence directe majeure est la forte diminution des blocages du noyau et des erreurs perçues au niveau de la charge de travail.

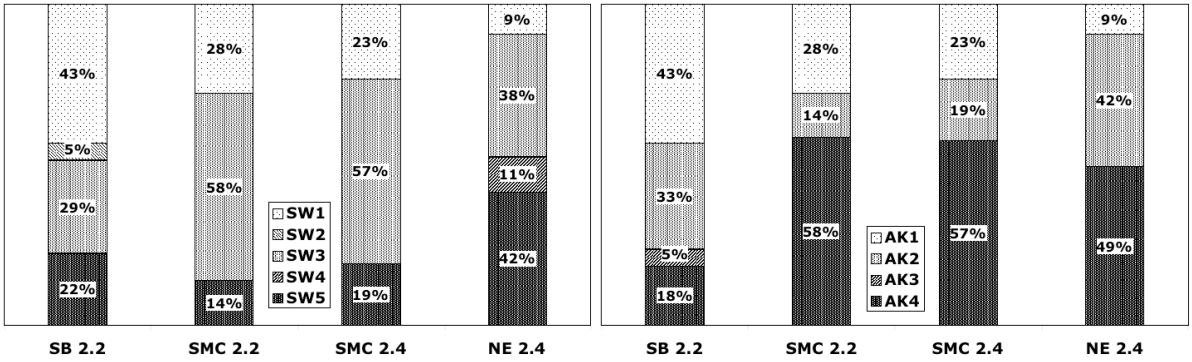
L'analyse avec priorité au premier événement collecté met en évidence que lorsqu'un événement de type « terminaison incorrecte » est observé en premier signifie qu'aucune autre notification n'a été relevée. Il est généralement le seul événement collecté lors de l'expérience, à moins qu'un blocage noyau soit relevé après la fin de la charge de travail (de tels cas sont rares). Dans la pratique, une analyse plus détaillée des données rassemblées est nécessaire pour s'assurer de ces relations. Nous verrons cela dans le cadre de l'analyse détaillée présentée à la fin de chapitre.

4.4.2 Points de vue et interprétation

Dans ce paragraphe, nous revisitons les premières analyses faites pendant les expériences entreprises à la lumière des trois points de vue définis au paragraphe 3.4.2. La Figure 4-7 récapitule les mesures correspondantes appliquées aux quatre séries d'expériences rapportées.



a) N` tificati` n d'événements par le système



b) Sûreté de la charge de travail **c) Disponibilité du noyau**

Figure 4-7 Interprétation des résultats selon différents points de vue

Comme cela est présenté dans le paragraphe 3.4, la gravité des événements observés dépend du contexte dans lequel on l'évalue. Dans un système critique comme le système informatique des commandes de vol d'un avion, une interruption du système a des conséquences graves. Le

critère principal est la disponibilité du système global. A contrario, si l'on considère maintenant un distributeur de billet, on préfèrera que celui-ci s'arrête complètement de fonctionner plutôt que d'agir de manière imprévisible. Le critère principal est alors la sûreté de résultats fournis par l'application, y compris au détriment de l'état général du système dans sa globalité.

Dans chaque cas, les pourcentages des diverses catégories de chaque mesure sont détaillés. Les catégories correspondant aux résultats les plus positifs apparaissent dans la partie supérieure des histogrammes en gris clair tandis que les plus critiques apparaissent au bas avec une nuance plus foncée. Nous considérons successivement l'ensemble des points de vue définis dans le chapitre trois.

4.4.2.1 Notifications d'événements

Considérons d'abord le point de vue « réponse du noyau » (*RK*). Dans ce cas, nous supposons que les expériences correspondant aux catégories *RK1* et *RK2* forment les résultats positifs de ce point de vue.

La distribution observée pour le pilote *SB 2.2* indique un comportement très positif: 84% des résultats observés indiquent une notification d'erreur (*RK1*=51% + *RK2*=33%). Cependant, parmi les 16% de résultats pour lesquels on a observé un mode de défaillance sans notification préalable d'erreur, un pourcentage significatif des résultats (plus de 2/3) correspondent aux événements *WI* (*RK5*=11%). Le tiers restant est dominé par des événements de type « blocage noyau » (*KH*) et, dans une moindre mesure, des interruptions de la charge de travail (*WA*). Il est important de remarquer que c'est la seule campagne d'expériences pour laquelle des événements non signalés de *WA* ont été diagnostiqués.

Les pilotes de carte réseau n'indiquent pas un comportement aussi positif. Par exemple, la distribution des catégories correspondant à la notification d'événements pour le pilote *SMC 2.2* indique que 41% des résultats observés correspondent à l'observation de défaillances sans notification préalable. Cependant, ceci est principalement dû aux événements de type « blocage noyau » (*KH*), la catégorie *RK4* correspondant à 29% des cas et à la catégorie *RK5* (événements *WI*) qui s'élève à 12%. La dernière catégorie, *RK3*, correspond aux interruptions non signalées de la charge de travail. Cette catégorie est souvent incluse dans *RK4* car une interruption de la charge de travail est souvent liée à un blocage du noyau..

La distributions des résultats présentés pour le pilote *SMC 2.4* montre que le portage du pilote sur la version 2.4 du noyau de *Linux* a sensiblement amélioré son comportement, le matériel étant resté identique d'une version à l'autre. Les catégories des modes de défaillance sans notification préalable sont réduites de 31% pour la précédente version (*SMC 2.2*) à 17% pour la nouvelle. Cette baisse correspond à la notification d'erreurs précédent aux événements de blocage du noyau ou de terminaison incorrecte de la charge de travail. De manière générale, environ 77% (60% + 17%) des modes de défaillance observés sont précédés par une notification d'erreur. Ceci confirme les observations déjà effectuées sur la base de l'analyse montrant les distributions du premier événement rassemblé.

4.4.2.2 Sûreté de la charge de travail

Comme nous l'avons présenté au paragraphe 3.4, du point de vue de la sûreté de la charge de travail, le principal objectif est d'éviter la livraison de résultats incorrects. Par conséquent, les catégories *SW4* et *SW5* sont considérées comme les plus négatives. Les résultats affichés par le pilote *SB 2.2* suggèrent un comportement beaucoup moins positif que ce qui a été déduit de l'analyse des résultats du point de vue de la notification d'événements: En effet, les cas les plus graves correspondants à la catégorie *SW5* (*WI* et aucun *KH*) représente 22% !

En outre, il faut noter que l'amélioration significative observée en ce qui concerne la notification d'erreur entre *SMC 2.2* et *SMC 2.4* ne se concrétise pas au niveau du comportement de la charge de travail. On n'observe pas de réduction sensible des événements *WI*. Ceci peut être expliqué par le fait que la plupart des notifications d'erreurs supplémentaires relevées dans *SMC 2.4* correspondent à des exceptions, plutôt qu'à des codes retour d'erreur (voir la Figue 4-6). En effet, ces exceptions signalent un comportement incorrect du système ayant une gravité importante sans pour cela être à même de lancer une procédure de recouvrement ou de récupération. L'erreur est signalée, mais elle n'est pas traitée.

L'analyse du point de vue de la sûreté de la charge de travail révèle un comportement similaire à celle effectuée selon le point de vue de la notification d'événements par le noyau: dans 53% des cas (*SW5*=42% + *SW4*=11%) les résultats observés comprennent un événement de *WI*, le plus grave.

4.4.2.3 Disponibilité du noyau

Pour le point de vue de disponibilité du noyau (*AK*), l'observation considérée comme la plus négative est caractérisée par un événement de blocage (*KH)*, car il a un impact déterminant sur la capacité du système à continuer à fournir ses services. Les catégories *AK3* et *AK4* sont considérées comme les plus critiques.

Les résultats obtenus pour le pilote *SB 2.2* indiquent que les fautes ont un impact significatif en ce qui concerne la disponibilité. En effet, on observe un événement de blocage du noyau dans 23% (18% + 5%) des cas. La terminaison incorrecte de la charge de travail représente 78% de ces cas (18% de 23%), ce qui indique que le noyau conserve certains services mais n'est plus en mesure de fournir ceux qui sont nécessaires au bon fonctionnement des applications.

Les résultats prouvent également que les fautes associées aux pilotes de carte réseau ont un impact très significatif sur l'ensemble du système: dans environ 50% des cas, les expériences d'injection de fautes provoquent un blocage du noyau. Pour les pilotes *SMC*, les résultats indiquent que les changements faits entre les deux versions n'ont aucun impact du point de vue de disponibilité du système.

4.4.2.4 Synthèse

La figure 4-8 présente une comparaison des faiblesses des pilotes de cartes réseaux testés. Sur une même version de noyau, il est intéressant d'observer des comportements différents entre deux implémentations d'une tâche similaire : Le pilote *Ne 2000* présente un meilleur comportement que son homologue *SMC* en termes de disponibilité du noyau (49% de défaillances entraînant une indisponibilité contre 57%). Mais les conclusions sont radicalement

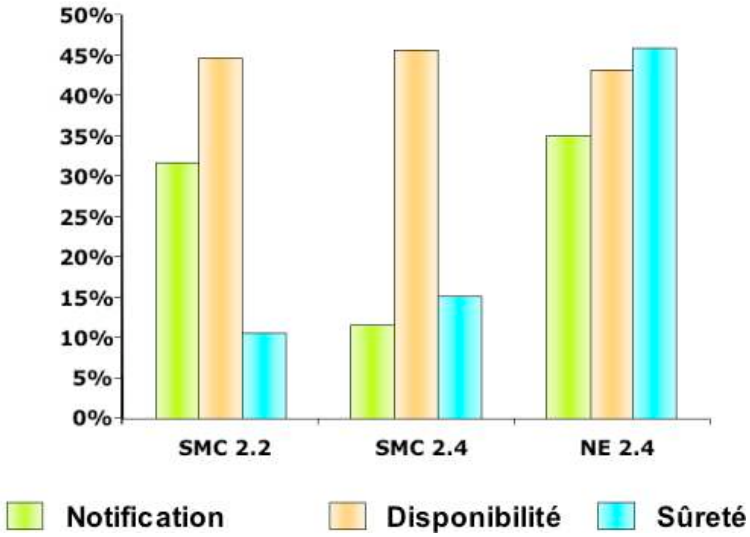


Figure 4-8 Comparaisons des faiblesses des pilotes de carte réseau

opposées en ce qui concerne les deux autres points de vue. Pour la notification d'événements (40% contre 17%) et pour la sûreté de la charge de travail (53% contre 19%), le pilote SMC révèle un meilleur comportement. Ceci s'explique par des choix d'implémentation différents.

Comme cela à déjà été souligné, la figure met en évidence que l'amélioration de la notification d'erreurs entre les deux versions (*SMC 2.2* et *SMC 2.4*) n'entraîne pas forcément une amélioration au niveau de l'occurrence des défaillances. En effet, il apparaît que la couverture des mécanismes de détection d'erreurs ont été améliorés d'une version à l'autre sans que les mécanismes de tolérance et de prévention de fautes n'y aient été associés.

4.4.3 Analyse détaillée

Comme nous l'avons indiqué dans le chapitre 2, l'évaluation de la robustesse d'une interface a pour objectif principal de permettre la comparaison du comportement de deux interfaces différentes. Cependant, lors de l'étude d'une interface, nous pouvons également réaliser une analyse plus détaillée du système en présence de fautes. L'objectif est alors d'expliquer de façon détaillée le comportement du système en présence de fautes afin, par exemple, de mettre en place des mécanismes de protection appropriés.

La répartition des défaillances par fonctions permet de décrire le comportement de l'ensemble des éléments de la DPI du noyau ciblé. À partir de cette répartition, les éléments les plus remarquables peuvent être analysés en profondeur, dans l'objectif de comprendre les mécanismes liés aux fonctions du noyau ou ceux du pilote de périphérique. L'impact des modifications du logiciel peut également être mesuré pour établir la progression en termes de sûreté de fonctionnement des composants concernés.

4.4.3.1 Répartition des défaillances par fonctions

Les expériences menées sur le noyau *Linux* permettent de mettre en évidence un canal de propagation d'erreurs des pilotes vers le noyau. En effet, une défaillance dans un pilote peut avoir un impact significatif sur la sûreté de fonctionnement du système d'exploitation. Une analyse des défaillances observées par fonctions (voir Figure 4-9) permet de révéler les composants de l'interface ayant un fort impact sur la sûreté de fonctionnement du système en

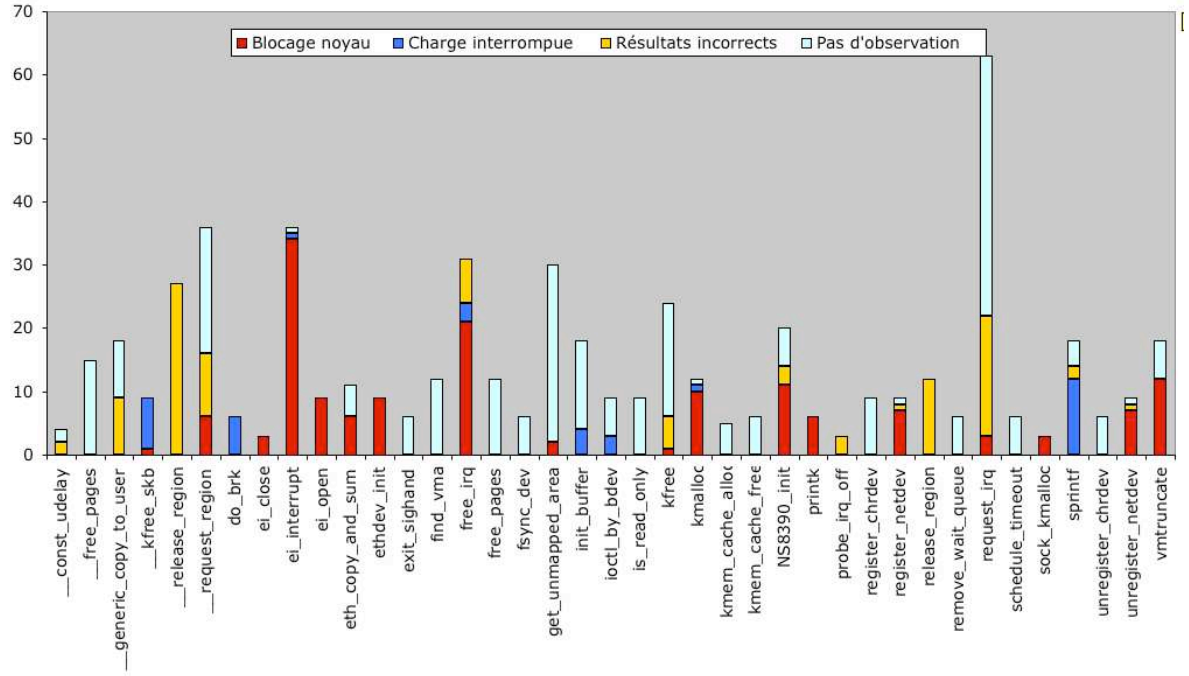


Figure 4-9 Répartition des défaillances par fonction (*Linux *2.4.18)

présence de fautes.

La catégorie « pas d'observation » regroupe toutes les expériences où aucune défaillance n'a été observée durant l'expérience d'injection de faute dans les paramètres de la fonction (en abscisse). La somme des 4 catégories représente le nombre d'injection de fautes qui ont été réalisées sur chaque fonction. Le nombre de corruption de fautes réalisables sur une fonction est directement liés aux nombres de paramètres de la fonction et de leur type. Ainsi, la fonction *request_irq* dispose d'une signature comportant un grand nombre de paramètres. À l'inverse, la fonction *ei_close* dispose d'un seul paramètre dans sa signature. D'une manière générale, on peut s'apercevoir que l'impact sur le système d'une injection de fautes varie de manière importante entre les fonctions.

Par exemple, la fonction *ei_interrupt*, chargé du traitement des interruptions (module réseau 8390), révèle un comportement très sensible aux fautes injectées. En effet, suite à une quarantaine de corruptions effectuées sur ses paramètres, nous avons pu observer dans la plupart des cas une défaillance de type blocage du noyau à partir de la machine observatrice. La remise en opération du système a donc demandé une intervention spécifique. Deux

corruptions ont provoqué l'interruption de la charge de travail. Enfin, seulement une expérience s'est terminée sans l'occurrence d'une défaillance.

Inversement, la fonction *free_pages*, chargée de la libération des pages mémoires non utilisées, a un comportement complètement différent. La quinzaine d'injections de fautes effectuées sur cette fonction n'ont provoqué aucune observation d'une défaillance. Du point de vue de la disponibilité du système, ce comportement peut être considéré comme très positif, et il est principalement dû à une double vérification effectuée en début du code correspondant à la fonction, vérifiant si la page n'est pas utilisée ou en attente d'utilisation.

La fonction *release_region*, chargée de libérer une région mémoire utilisée, a un comportement particulièrement négatif du point de vue de la sûreté des résultats fournis par la charge de travail. En effet, sur les 27 expériences d'injection de fautes menées sur ce symbole, nous avons pu observer la production de résultats incorrects par la charge de travail. La fonction *release_region* ne peut effectuer des vérifications du même type que la fonction *free_pages* avant la réalisation de la libéralisation de la mémoire. En effet, son rôle est de libérer un espace mémoire utilisé et occupé par un processus du système.

4.4.3.2 Critères de sûreté de fonctionnement et de performance

La fonction *eth_copy_and_sum* présente dans la DPI de la version 2.2.20 de *Linux* provoque de nombreux blocages du noyau. Cette fonction permet de copier des données provenant d'un bloc mémoire d'un périphérique Ethernet dans une mémoire tampon du noyau lors d'une opération arithmétique. Son code se compose d'une partie originale et d'un appel à la fonction *mem_cpy*. Ainsi, aux faiblesses correspondant à son propre code peuvent s'ajouter celles de la fonction *mem_cpy*. Les versions des pilotes testées sur la version 2.4.18 n'utilisent plus *eth_copy_and_sum* de la même manière. En effet, elle n'est désormais plus utilisée dans les nouvelles versions des pilotes testés et ne se compose plus que du code de la nouvelle version de *mem_cpy*. Les résultats de *mem_cpy* étant positif du point de vue de la sûreté de fonctionnement (peu de défaillances), les résultats de *eth_copy_and_sum* sont désormais identique. L'amélioration de la sûreté de fonctionnement d'une interface comme la DPI passe par une réduction des fonctions disponibles pour réaliser une tâche similaire, en selectionnant celles révélant le meilleur comportement (sûreté de fonctionnement et performance). Par exemple, dans la version 2.2 de Linux, un pilote *ethernet* pouvait faire appel à différente fonctions pour transférer les données de la mémoire d'un périphérique. Dans la version 2.4, les fonctions de la DPI les moins sûres ont été « retirées » de la DPI.

Certaines fonctions du noyau sont volontairement épurées de toutes vérifications et de tests afin d'améliorer les performances du noyau. C'est le cas de la fonction *printk*. Elle sert à afficher un message à l'écran pour un processus en mode privilégié. Elle a subi de larges modifications entre les deux versions du noyau sans modifier son comportement vis-à-vis des paramètres invalides en entrées. Cependant cette fonction fréquemment utilisée est assez particulière. Les développeurs du noyau ont préféré conserver la dynamique de la fonction vu son niveau de complexité, un unique paramètre de type chaîne de caractère, et sa fréquence d'utilisation.

4.4.3.3 Choix des critères de sûreté de fonctionnement

La fonction *kmem_cache_free* est une fonction servant à libérer la mémoire utilisée par les objets *dentry* (objets utilisés par le système de fichier pour parcourir l'arborescence des répertoires). Elle a été totalement modifiée entre les deux versions du noyau et le résultat est mitigé. Tandis que les blocages du noyau et les interruptions de la charge de travail présent dans la version 2.2.20 n'apparaissent plus dans les campagnes menées dans la version 2.4 du noyau, la sûreté de la charge de travail a diminué par l'apparition de nombreux cas de terminaison incorrecte au niveau applicatif. L'amélioration d'un critère de sûreté de fonctionnement (comme ici la disponibilité) peut s'accompagner de la baisse d'un autre critère (comme la sûreté de la charge de travail). En effet, la capacité à juger positif le comportement du système en présence de fautes dépend généralement de son domaine d'application. L'analyse en tenant compte des points de vue de sûreté de fonctionnement prend toute son importance dans ce cas.

4.4.3.4 Amélioration de la sûreté de fonctionnement : le symbole *iget4*

Sous *Linux*, chaque fichier correspond à un ensemble de blocs disque et est associé à un ensemble de paramètres appelés *inode*. L'*inode* contient toutes les informations importantes sur le fichier comme sa taille, son propriétaire et ses droits d'accès. *iget4* est un symbole de la DPI qui a pour rôle d'aider le système de fichiers à gérer les *inodes*, principalement en maintenant l'intégrité de leur « cache ». Cette fonction est souvent mal comprise et revient régulièrement dans les forums de discussions des spécialistes.

Tout système de fichiers peut fournir sa propre fonction de lecture d'un *inode*. Dans cette fonction, il fera, a priori, un appel à la fonction *iget4* lorsqu'il souhaitera accéder à un *inode* particulier. Si *iget4* le trouve dans le cache, la fonction le retournera. Dans le cas contraire, la fonction va créer un nouvel *inode*, initialiser les champs de l'*inodes* et renvoyer le nouvel *inode* ainsi créé. *iget4* traite les verrous nécessaires pour s'assurer qu'aucun *inode* ne soit créé ou lu (par la fonction de lecture d'*inode*) en double.

Durant les expériences qui ont été effectuées sur la version 2.2.20 du noyau *Linux*, *iget* a révélé un comportement faible du point de vue de la disponibilité du noyau. En effet, dans de nombreux cas, nous avons observé un blocage du noyau. Ceci peut s'expliquer par une gestion incorrecte des verrous.

En pratique, la fonction *iget* fait appel à la fonction *iget4*. L'analyse du code du symbole *iget4* sur la Figure 4-10 montre comment la gestion des verrous pouvait être facilement perturbée.

La fonction *spin_lock* active le verrou du cache. La fonction *find_inode* peut alors effectuer la recherche de l'*inode* dans celui-ci. Si l'*inode* recherché est trouvé, le verrou sur le cache est désactivé, l'*inode* est placé dans une file d'attente et le traitement de la fonction est terminé.

Lors de l'exploitation de *GNU/Linux*, les relevés en opération ont révélé des faiblesses sur la gestion des verrous du cache d'*inodes* dans les versions antérieures à la version 2.4. Pour chacun, le système de fichiers (Coda, NFS ou ReiserFS) utilisait *iget4*. Les verrous étaient gérés à la fois par la fonction *iget4* et la fonction *get_new_inode*. Dans certains cas, le verrou sur le cache pouvait ne pas avoir été levé. Cela a pour effet d'empêcher le traitement ultérieur

sur le cache d'*inodes*. La fonction a été corrigée dans la version 2.4 en la modifiant pour limiter les blocages du système de fichier.

```
struct inode *iget4(struct super_block *sb, unsigned long ino,
find_inode_t find_actor, void *opaque){
        struct list_head * head = inode_hashtable + hash(sb,ino);
        struct inode * inode;
        spin_lock(&inode_lock);
        inode = find_inode(sb, ino, head, find_actor, opaque);
        if (inode) {
                spin_unlock(&inode_lock);
                wait_on_inode(inode);
                return inode;
        }
        /* get_new_inode() will do the right thing, releasing
        * the inode lock and re-trying the search in case it
        * had to block at any point.
        */
        return get_new_inode(sb, ino, head, find_actor, opaque); }
```

Version extraite du noyau *Linux* 2.2.20

```
struct inode *iget4(struct super_block *sb, unsigned long ino,
find_inode_t find_actor, void *opaque) {
        struct list_head * head = inode_hashtable + hash(sb,ino);
        struct inode * inode;
        spin_lock(&inode_lock);
        inode = find_inode(sb, ino, head, find_actor, opaque);
        if (inode) {
                __iget(inode);
                spin_unlock(&inode_lock);
                wait_on_inode(inode);
                return inode;
        }
        spin_unlock(&inode_lock);
        /* get_new_inode() will do the right thing, re-trying the
        * search in case it had to block at any point.
        */
        return get_new_inode(sb, ino, head, find_actor, opaque); }
```

Version extraite du noyau *Linux* 2.4.18

Figure 4-10 Code correspondant au symbole iget4

Les résultats issus des expériences menées sur la version 2.4 de la fonction *iget4* montrent un meilleur comportement du point de vue la disponibilité du noyau. En effet, les expériences ne relèvent aucun blocage noyau suite aux injections. Toutefois, il faut noter qu'une part des fautes qui provoquait un blocage noyau entraîne désormais une défaillance applicative (le fichier n'a pu être ouvert ou modifié).

4.5 Conclusion

La mise en œuvre expérimentale de l'approche présentée dans le chapitre trois a permis d'évaluer une grande partie de la DPI du noyau *Linux* face aux défaillances des pilotes de périphériques. Le choix du système *Linux* a été motivé pour sa reconnaissance dans le milieu, la disponibilité des informations concernant son code et le fort retour d'expérience dont bénéficie le noyau. La proposition présentée au chapitre 3 a pu être testée plus facilement.

L'étape d'analyse de l'interface, avec l'extraction des éléments qui la composent, est une tâche difficile pour l'évaluation de sa robustesse mais elle est néanmoins essentielle. Elle se compose pour une grande part de l'étude des fonctions de la DPI et de leurs signatures. En effet, la principale différence d'un système à l'autre est la composition de la DPI. Les types de données et leurs intervalles de validité sont sensiblement identiques d'un système à l'autre et sont fixés généralement par le langage de programmation.

La spécification du profil d'exécution dépend de cette analyse. En effet, une fois les éléments connus de l'interface, si le matériel le permet, des requêtes provenant de l'interface utilisateur doivent suffire à les activer. L'utilisation d'une charge générique, comme un étalon de performances, ne permet pas d'activer certaines fonctionnalités spécifiques du pilote de périphérique comme la charge et la décharge du module dans le noyau. De plus, l'utilisation d'une charge générique entraîne souvent des temps d'exécution plus importants qu'une charge dédié à un périphérique, pilote de périphérique ou service du noyau.

La plate-forme expérimentale a été présentée comme étant modulable selon l'objectif de la campagne d'injection de fautes. De plus, elle permet d'accueillir plusieurs campagnes simultanées. Elle est étudiée pour minimiser l'intervention manuelle et permettre de mener des expériences rapidement. Nous avons présenté dans ce chapitre l'outil mis en place pour mener des campagnes d'expériences sur *Linux*.

La décomposition de l'outil logiciel en plusieurs modules est destinée à favoriser la portabilité de l'outil. Une partie du logiciel est dépendante du système d'exploitation. Elle doit évoluer lorsque l'outil sera porté sur un noyau différent. Une seconde partie du logiciel est dépendante de l'architecture de la plate-forme matérielle de la machine (routines en langage Assembleur X86). Cette section de l'outil devra être modifiée lors du changement de machine cible. Enfin, la troisième partie est purement logicielle et son code est indépendant du système ou de la plate-forme.

Afin d'illustrer et évaluer notre approche, nous avons conduit une série d'expériences sur deux versions du noyau de *Linux* et avons examiné trois pilotes. Les campagnes d'expériences ont permis l'évaluation de près de 150 symboles, ce qui représente environ 30% de l'ensemble des fonctions du noyau, tous services confondus.

En pratique, davantage de symboles ont été étudiés, mais un grand nombre n'a pu être activé. La principale raison réside dans les limites matérielles. En effet, le nombre de périphériques sur les machines cibles est limité. Certains symboles et services du noyau sont utilisés pour des conditions assez spécifiques et ne correspondent pas à une utilisation courante d'un ordinateur personnel ou d'une station de travail. Dans une moindre mesure, la seconde raison peut être une faiblesse de la charge de travail qui n'a pu activer les symboles visés : Soit la charge de travail n'a pas activé la portion de code du pilote utilisé, soit la fonction du pilote n'est pas utilisée par le pilote pour l'architecture testée. Le taux de symboles évalués par rapport au nombre de symboles utilisés par le système lors de l'exécution de la charge de travail s'élève à plus de 75%. Le quart restant correspondant aux symboles qui n'ont pu être activés par la charge de travail mais qui sont présents dans les pilotes du système ciblé par le test.

Les expériences entreprises sur le pilote de *SMC* indiquent une amélioration significative entre les deux versions du noyau *Linux* considérées du point de vue de la détection des fautes, sans qu'il y ait eu de conséquences sur les modes de défaillances qu'elles engendrent. La progression, en termes de sûreté de fonctionnement au niveau de la signalisation d'une erreur n'a pas été suivie par une évolution des systèmes permettant le recouvrement de la faute par le système. Nous avons identifié un comportement légèrement meilleur concernant la disponibilité lors des expériences entreprises sur le pilote *Ne 2000* par rapport à son homologue *SMC*, contrairement à la sûreté de la charge de travail et la notification d'événements. Nous considérons l'observation d'une grande proportion de codes d'erreur (en particulier dans le cas du premier événement observé) comme un résultat positif lors de la caractérisation détaillée

des comportements incorrects induits par les paramètres corrompus. En outre, ces codes forment une base utile sur laquelle la gestion d'erreur spécifique pourrait être mise en application.

Nous avons complété ces analyses par une étude détaillée du comportement de certains composants de l'interface DPI de *Linux*. Les résultats montrent que l'amélioration par la mise en place de mécanisme de détection d'erreurs sur l'interface change le comportement du système vis-à-vis de la sûreté de fonctionnement. Cependant, les modifications ne sont pas toujours réalisables sans affaiblissement des performances actuelles du noyau. De plus, le gain sur un critère de sûreté de fonctionnement comme la sûreté des résultats peut, quelquefois, se faire au détriment d'un autre critère comme la disponibilité. La prise en compte des besoins de sûreté de fonctionnement du domaine est, plus que jamais, essentielle dans l'évaluation d'un composant en présence de fautes. Ces résultats obtenus valident l'intérêt et la viabilité de la méthodologie proposée.

Conclusion générale

Les travaux présentés dans ce mémoire ont été motivés par un constat simple : les pilotes de périphérique sont la principale cause des défaillances des systèmes d'exploitation modernes. Les connaissances nécessaires à leur mise en œuvre, la complexité croissante des interactions entre les dispositifs interconnectés, les contraintes de développement et les problèmes économiques associés affectent la qualité de ces logiciels et leur comportement en opération. Ils constituent actuellement un des enjeux les plus importants de la sûreté de fonctionnement des systèmes d'exploitation. Le noyau du système d'exploitation et le matériel ne sont plus considérés comme la source principale des dysfonctionnements observés en opération et l'attention se porte désormais sur les extensions du noyau, et particulièrement les pilotes de périphériques.

Les concepteurs de systèmes informatiques cherchent actuellement des solutions pour :

- Résoudre les problèmes de sûreté de fonctionnement du point de vue des pilotes et évaluer les propositions effectuées.
- Évaluer la sûreté de fonctionnement d'un système d'exploitation COTS avant son intégration en tant que composant au sein d'un système informatique critique.
- Analyser et comparer différentes alternatives possibles de COTS candidats.

Les techniques d'étalonnage de la sûreté de fonctionnement permettent de fournir des moyens génériques et reproductibles pour la caractérisation du comportement des systèmes en présence de fautes. Un étalon de sûreté de fonctionnement se distingue des techniques d'évaluation et de validation existantes par l'accord qu'il doit assurer entre les différents acteurs impliqués. Les mesures, les moyens de les obtenir et leur domaine de validité sont spécifiés à travers cet accord et imposent des règles précises lors de la définition de la méthode d'évaluation.

Dans ce sens, nous avons développé une méthode pour l'évaluation de la robustesse des noyaux face aux comportements anormaux des pilotes de périphériques. Après avoir analysé et précisé les caractéristiques de l'interface pilote-noyau, nous avons proposé une méthode originale d'injection de fautes basée sur la corruption des paramètres des fonctions propres au noyau manipulées au niveau de cette interface. Cette approche a été motivée par sa portabilité, la nature des erreurs provoquées et leur représentativité vis-à-vis des conséquences des fautes réelles.

Toutefois, les résultats obtenus par une technique d'évaluation de la sûreté de fonctionnement dépendent de nombreux critères. Afin d'exploiter au mieux les résultats fournis par notre proposition, nous avons proposé un cadre d'analyse multicritères pour l'interprétation des

résultats en fonction de divers points de vue de sûreté de fonctionnement. Ces points de vue permettent d'exprimer la capacité du système évalué à mettre en œuvre des mécanismes de tolérance aux fautes. Nous avons ainsi défini trois classes de mesures de la sûreté de fonctionnement : la notification d'erreur par le noyau, la disponibilité du noyau et la sécurité de la charge de travail. La première se focalise sur la capacité du noyau à signaler les comportements anormaux. La deuxième se concentre sur la capacité du système à assurer la continuité de son service. Enfin, la troisième s'intéresse au maintien d'un service correct des applications. Ces classes fournissent des moyens pratiques pour l'analyse d'un noyau selon ses besoins de sûreté de fonctionnement, par exemple lors de l'intégration dans un système embarqué critique.

Le cadre générique présenté dans ce manuscrit permet de définir des étalons de sûreté de fonctionnement adaptés à l'analyse des systèmes d'exploitation en présence de pilotes défaillants. Nous avons illustré la méthode proposée en la mettant en œuvre dans le cas de *Linux*. L'étape d'analyse de l'interface, avec l'extraction des éléments qui la composent, constitue une des tâches centrales pour l'évaluation de sa robustesse. La spécification du profil d'exécution découle de cette analyse. La plateforme expérimentale et la technique d'observation du système cible se présentent sous la forme d'un logiciel modulable. Cette modularité favorise significativement l'adaptation et la portabilité de l'outil à de nouvelles cibles. En effet, une partie de ce logiciel est dépendante du système d'exploitation et doit être adaptée selon le noyau ciblé. Une autre partie du logiciel est dépendante de l'architecture de la plateforme matérielle de la machine (routines assembleurs). Elle peut nécessiter des modifications lors du changement du support matériel. La troisième partie, composée essentiellement des charges de travail et de fautes, est indépendante du système ou de la plateforme.

Les résultats obtenus concernent principalement deux aspects :

- Le comportement du noyau vis-à-vis d'un environnement défaillant et l'impact des améliorations effectuées à travers deux versions.
- La comparaison des comportements de différents composants logiciels d'un périphérique similaire.

Une partie des campagnes d'expériences présentées dans ce manuscrit concerne l'évolution du noyau *Linux* à travers deux versions stables. L'évaluation de la version 2.2 de Linux révèle le fort impact des fautes injectées. Un comportement anormal des pilotes de périphérique provoque de nombreuses perturbations dans le système, sous la forme de blocages du noyau ou de comportements anormaux de la charge de travail. Certaines fonctions sont plus sensibles que d'autres, c'est notamment le cas des fonctions liées à la gestion des interruptions. Il est important de noter que certaines fonctions sont volontairement « allégées » dans le but d'améliorer les performances du système. L'équilibre entre performance et sûreté de fonctionnement se situe aujourd'hui au centre des préoccupations des concepteurs et des intégrateurs de noyaux pour les systèmes critiques. De plus, la mise en évidence des faiblesses de l'interface pilote noyau de *Linux* permet de focaliser les efforts pour l'amélioration de la sûreté de fonctionnement. Enfin, l'analyse approfondie des modifications effectuées sur la seconde version de *Linux* ciblée (2.4) montre que l'impact des évolutions n'est pas toujours bénéfique et dépend de la classe de mesure considérée.

L'intérêt des classes de mesures proposées a également été mis en évidence pour la comparaison des résultats fournis lors des campagnes d'expériences visant les fonctions utilisées par les pilotes de deux cartes réseaux (*Ne2000* et *SMC-Ultra*), dans le cas des deux versions du noyau *Linux* ciblées. Le comportement observé pour chaque pilote dépend étroitement de la classe de mesure considérée. En fonction du besoin de l'utilisateur, les résultats obtenus constituent alors un support objectif quant au choix du meilleur couple pilote/version en terme de sûreté de fonctionnement.

De plus, les expériences réalisées sur le pilote de la carte réseau *SMC-Ultra* indique une amélioration significative entre les deux versions du noyau *Linux* considérées du point de vue de la notification d'événements, sans qu'il y ait eu de conséquences sur les modes de défaillances observées. Cela résulte de ce que la signalisation des erreurs n'est pas suivie par une prise en compte par des mécanismes de recouvrement. Cependant, ces manifestations forment une base utile pour améliorer la gestion d'erreur spécifique.

La sûreté de fonctionnement liée aux pilotes représente aujourd'hui un enjeu considérable dans l'élaboration des systèmes informatiques. La méthode présentée dans ce manuscrit permet d'améliorer la sûreté de fonctionnement des systèmes informatiques lors de trois étapes distinctes :

- Le processus de développement des noyaux : Evaluation et spécification d'une interface jusque-là souvent négligée, caractérisation du système face aux faiblesses observées permettant de faciliter leurs identifications pour son amélioration.
- La conception et le développement des pilotes : Evaluation de l'impact des défaillances des pilotes de périphériques produits. Elle permet également d'évaluer les mécanismes d'amélioration de la sûreté de fonctionnement des pilotes de périphérique mis en œuvre.
- La phase d'intégration : Evaluation et comparaison de différentes alternatives de pilotes et de noyaux en fonction des critères de sûreté de fonctionnement recherchés.

Les travaux expérimentaux présentés dans ce mémoire traitent de plusieurs versions d'un même système d'exploitation. Il faudra approfondir les travaux sur *Linux* en activant une plus grande partie de l'interface et en enrichissant la charge de fautes, la charge de travail et en prenant en compte les données temporelles. Cependant, l'élargissement à des logiciels d'exploitation alternatifs ayant mis en place des améliorations du point de vue de la sûreté de fonctionnement est nécessaire. Nos travaux ont porté également sur le système d'exploitation *Windows XP* où la méthodologie peut être mise en œuvre aisément grâce à une architecture logicielle structurée. L'objectif de ce portage serait principalement de mesurer l'impact des environnements de développement et d'exécution sur le comportement du noyau vis-à-vis des défaillances des logiciels de pilotage de périphériques. Plus généralement, les noyaux qui devront être étudiés seront ceux ayant mis en place des améliorations de la sûreté de fonctionnement : au niveau de l'architecture, de la conception, du développement, ou encore de l'exécution du code.

Par exemple, la mise en œuvre de notre approche sur des systèmes récents, tel GNU/*Hurd*, désormais basé sur le micronoyau *L4*, permettrait une comparaison intéressante de l'évolution

de la propagation des fautes dans les systèmes d'exploitation modernes (pilotes au niveau applicatif). L'étude des noyaux *Linux-Nooks* et *Linux-SD* permettrait quant à elle d'évaluer l'efficacité de l'approche par recouvrement logiciel. Les noyaux QNX, RT-*Linux* et *Windows CE/XPE* constituent également des cibles potentielles intéressantes dans le domaines des systèmes embarqués.

La comparaison du comportement de ces systèmes en présence de fautes fournirait des éléments de réponse objectifs aux besoins des intégrateurs de COTS et évaluerait l'apport des approches suivies ces dernières années pour l'amélioration de la sûreté de fonctionnement du point de vue des pilotes de périphériques.

Références bibliographiques

[Albinet *et al.* 2004] A. Albinet, J. Arlat, J.-C. Fabre, "Characterization of the Impact of Faulty Drivers on the Robustness of the *Linux* Kernel," *Proceedings International Conference on Dependable Systems and Networks (DSN-2004)*, (Florence, Italy), 2004, pp. 867-876.

[Albinet 2003] A.Albinet, « Sûreté de fonctionnement des systèmes d'exploitation. Influence des pilotes de périphérique », 4ème Congrès de l'Ecole Doctorale Systèmes (EDSYS'2003), (Toulouse, France), 2003.

[Arlat *et al.* 1990] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E. Martins et D. Powell, "Fault Injection for Dependability Validation — A Methodology and Some Applications", *IEEE Transactions on Software Engineering*, 16 (2), pp.166-182, 1990.

[Arlat *et al.* 1993] J. Arlat, A. Costes, Y. Crouzet, J.-C. Laprie et D. Powell, "Fault Injection and Dependability Evaluation of Fault-Tolerant Systems", *IEEE Transactions on Computers*, 42 (8), pp.913-923, 1993.

[Arlat & Crouzet 2002] J. Arlat et Y. Crouzet, "Faultload Representativeness for Dependability Benchmarking", in *Supplement Proceedings International Conference on Dependable Systems and Networks (DSN-2002),* (Washignton, D.C., USA), pp.F29-F30, 2002.

[Bach 1987] M. J. Bach, *The Design of the UNIX Operating System,* Englewood Cliffs, NJ, USA, 1987.

[Béounes *et al.* 1993] C. Béounes, M. Aguéra, J. Arlat, C. Bourdeau, J.-E. Doucet, K. Kanoun, J.-C. Laprie, S. Metge, J. Moreira de Souza, D. Powell et P. Spiesser, "SURF-2: A Program for Dependability Evaluation of Complex Hardware and Software Systems", in *Proceedings 23rd International Symposium on Fault-Tolerant Computing (FTCS-23),* (Toulouse, France), pp.668-673, IEEE CS Press, 1993.

[Bershad *et al.* 1995] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers et S. Eggers, "Extensability, safety and performance inthe SPIN operating system", in Proceedings *14th Symposium on Operating Systems Principles,* (CooperMontain, Colorado), pp.267-284, 1995.

[Bondavalli *et al.* 2000] A. Bondavalli, I. Mura, S. Chiaradonna, R. Filippini, S. Poli et F. Sandrini, "DEEM: a Tool for the Dependability Modeling and evaluation of Multiple Phased Systems", in *Proceedings International Conference on Dependable Systems and Networks (DSN-2000),* (New York, NY, USA), pp.231-236, IEEE CS Press, 2000.

[Borrel 1996] M. Borrel, *Interaction entre composants matériels et logiciels de systèmes tolérants aux fautes: caractérisation, formalisation et modélisation - Application à la sûreté de fonctionnement du Cautra,* Rapport LAAS n° 96001, Institut National Polytechnique de Toulouse, 1996.

[Bovet & Cesati 2000] D. P. Bovet et M. Cesati, *Understanding the Linux Kernel,* O'Reilly (ISBN : 0-596-00002-2), 2001.

[Brown & Seltzer 1997] A. Brown et M. Seltzer, "Operating System Benchmarking in the wake of Lmbench: A Case Study of the Performance of NetBSD on the Intel x86 Architecture", in *Proceedings of the 1997 ACM SIGMETRICS Conference on the Measurements and Modeling of Computer Systems,* (Seattle, WA, USA), 1997.

[Cant 1999] C. Cant, *"Writing Windows WDM Device Drivers",* CMP Books, 1999.

[Carreira *et al.* 1998] J. Carreira, H. Madeira et J. G. Silva, "Xception: A Technique for the Experimental Evaluation of Dependability in Modern Computers", *IEEE Transactions on Software Engineering*, 24 (2), pp.125-136, 1998.

[Cheynet *et al.* 2000] P. Cheynet, B. Nicolescu, R. Velazco, M. Rebaudengo, M. Sonza Reorda et M. Violante, "Experimentally Evaluating an Automatic Approach for Generating Safety-Critical Software with respect to Transient Errors", *IEEE Transactions on Nuclear Science*, 47 (6), pp.2231-2236, 2000.

[Chevochot & Puaut 2001] P. Chevochot and I. Puaut, "Experimental Evaluation of the Fail-Silent Behavior of a Distributed Real-Time Run-Time Support built from COTS Components", *in Proceeding of the International Conference on Dependable Systems and Networks (DSN-2001),* (Göteborg, Sweden), 2001.

[Chillarege *et al.* 1992] R. Chillarege, I. S. Bhandari, J. K. Chaar, M. J. Halliday, D. S. Moebus, B. K. Ray et M. Y. Wong, "Orthogonal Defect Classification - A Concept for In-Process Measurements", *IEEE Transactions of Software Engineering*, 18 (11), pp.943-956, 1992.

[Chiueh *et al* 1999] T. Chiueh, G. Venkitachalam et P. Pradhan, "Integrating segmentation and paging protection for safe, efficient and transparent software extensions", in *17th ACM Symposium on Operating Systems Principles (SOSP-99),34(5)*, pp.140-153, ACM Press, 1999.

[Chou *et al.* 2001] A. Chou, J. Yang, B. Chelf, S. Hallem et D. Engler, "An Empirical Study of Operating Systems Errors", in *Proceedings 18th ACM Symposium on Operating Systems Principles (SOSP-2001),* (Banff, AL, Canada), pp.73-88, ACM Press, 2001.

[Christmansson & Chillarege 1996] J. Christmansson et R. Chillarege, "Generation of an Error Set that Emulates Software Faults-Based on Field Data", in Proceedings *26th International Symposium on Fault Tolerant Computing (FTCS-26),* (Sendai, Japan), 1996.

[Christmansson *et al.* 1998] J. Christmansson, M. Hiller et M. Rimen, "An Experimental Comparison of Fault and Error Injection", in Proceedings *9th International Symposium on Software Reliability Engineering,* (Paderborn, Germany), pp.369-378, 1998.

[Ciardo *et al*. 1989] G. Ciardo, J. K. Muppala et K. S. Trivedi, "SPNP: Stochastic Petri Net Package", in *Proceedings 3rd International Workshop on Petri Nets and Performance Models,* pp.142-151, IEEE CS Press, 1989.

[Costa *et al*. 2001] D. Costa, T. Rilho et H. Madeira, "ESFFI - A Novel Technique for the Emulation of Software Faults in COTS Components", in Proceedings *8th Annual IEEE International Conference on the Engeneering of Computer-Based Systems (ECBS 2001),* (Washignton D.C., USA), 2001.

[Czeck 1993] E. W. Czeck, "Estimates of the Abilities of Software-Implemented Faulkt Injection to Represent Gate-Level Faults", in Proceedings *International Workshop on Fault and Error Injection for Dependability Validation of Computer Systems,* (Gothemburg, Suisse), 1993.

[Duraes & Madeira 2002] J. Durães, H. Madeira, "Emulation of Software Faults by Selective Mutations at Machine-code Level," in *Proceedings (ISSRE-2002)*, (Annapolis, MD, USA), 2002, pp. 329-340.

[Duraes & Madeira 2003] J. Durães, H. Madeira, "Mutidimensional Characterization of the Impact of Faulty Drivers on the Operating Systems Behavior," IEICE *Trans. on Information and Systems*, vol. E86-D, no. 12, pp. 2563-2570, 2003.

[Edwards & Matassa 2002] D. Edwards et L. Matassa, "An Approach to Injecting Faults into Hardened Software," Proceedinds *on Ottawa Linux Symposium (OLS-2002),* (Ottawa, ON, Canada), 2002, pp. 146-17

[Engler *et al*. 1995] D. Engler, M. F. Kaashoek et J. O'Toole, "Exokernel: An Operating System Architecture for Application-Level Resource Management", in Proceedings *Symposium on Operating Systems Principles,* (Colorado, USA), pp.251-266, 1995.

[Fassino *et al*. 2002] J.-P. Fassino, J.-B. Stefani, J. Lawall et G. Muller, "THINK: A Software Framework for Component-based Operating System Kernels", in *Proceedings of Usenix Annual Technical Conference,* (Monterey, USA), 2002.

[Fuchs 1996] E. Fuchs, "An Evaluation of the error Detection Mechanisms in MARS Using Software Implemented Fault Injection", in Proceedings *European Dependable Computing Conference (EDCC-1),* (Toarmina, Italy), pp.73-90, 1996.

[Fuchs 1998] E. Fuchs, "Validating the Fail-Silence of the MARS Architecture", in *Dependable Computing for Critical Applications (Proceedings 6th IFIP International Working Conference on Dependable Computing for Critical Applications: DCCA-6, Grainau, Germany* (M. Dal Cin, C. Meadows and W. H. Sanders, Eds.), Dependable Computing and Fault-Tolerant Systems, 11, (A. Avizienis, H. Kopetz and J.-C. Laprie, Eds.), pp.225-247, IEEE CS Press, Los Vaqueros, CA, USA, 1998.

[Gu *et al*. 2003] W. Gu, Z. Kalbarczyk, R. K. Iyer, Z. Yang, "Characterization of Linux Kernel Behavior under Errors," in *Proceedings International Conference on Dependable Systems and Networks (DSN-2003),* (San Francisco, CA, USA), 2003, pp. 459-468.

[Gu *et al*. 2004] W. Gu, Z. Kalbarczyk, R. K. Iyer, "*Error sensitivity of the Linux KernelExecuting on PowerPC G4 and Pentium 4 Processors,*" in *Proceedings International*

Conference on Dependable Systems and Networks (DSN-2004), (Florence, Italie), 2004, pp. 887-896.

[Härtig *et al.* 1997] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg et J. Wolter, "The Performance of µ-kernel-based systems", in Proceedings *16th ACM Symposium on Operating System Principles (SOSP'97),* (Saint-Malo, France), 1997.

[Hildebrand 1992] D. Hildebrand, "An Architectural Overview of QNX", in *Proceedings of the USENIX Workshop on Micro-kernels and Other Kernel Architectures,* (Seattle, WA, USA), pp.113-126, 1992.

[Hiller *et al.* 2001] M. Hiller, A. Jhumka et N. Suri, "An Approach for Analysing the Propagation of Data Errors in Software", in *Proceedings International Conference on Dependable Systems and Networks (DSN-2001),* (Göteborg, Sweden), pp.161-170, IEEE CS Press, 2001.

[Jarboui *et al.* 2002a] T. Jarboui, J. Arlat, Y. Crouzet, K. Kanoun et T. Marteau, "Analysis of the Effects of Real and Injected Software Faults: Linux as a Case Study", in *Proceedings 2002 Pacific Rim International Symposium on Dependable Computing (PRDC 2002),* (Tsukuba, Japan), IEEE CS Press, 2002a.

[Jarboui *et al.* 2002b] T. Jarboui, A. Kalakech et O. Guitton, "Assessment of the robustness of Windows2000 via the injection of select bit-flips", in Proceedings *4th European Dependable Computing Conference (EDCC-4). Fast abstracts track,* (Toulouse, France), pp.9-10, 2002b.

[Jarboui *et al.* 2003] T. Jarboui, J. Arlat, Y. Crouzet, K. Kanoun, T. Marteau, "Impact of Internal and External Software Faults on the Linux Kernel," IEICE Trans. Inf. & Syst., vol. E86-D, no. 12, pp. 2571-2578, December 2003.

[Kalakech *et al.* 2004] A. Kalakech, K. Kanoun, Y. Crouzet, J. Arlat, "Benchmarking the Dependability of Windows NT4, 2000 and XP," *Proceedings International Conference on Dependable Systems and Networks (DSN-2004)*, (Florence, Italy), 2004.

[Kanawati *et al.* 1995] G. A. Kanawati, N. A. Kanawati et J. A. Abraham, "FERRARI: A Flexible Software-Based Fault and Error Injection System", *IEEE Transactions on Computers*, 44 (2), pp.248-260, février 1995.

[Kanoun *et al.* 1999] K. Kanoun, M. Borrel, T. Moreteveille et A. Peytavin, "Modeling the Dependability of CAUTRA, a Subset of the French Air Traffic Control System", *IEEE Transactions on Computers*, 48 (5), pp.528-535, 1999.

[Kanoun *et al.* 2002] K. Kanoun, H. Madeira et J. Arlat, "A Framework for Dependability Benchmarking", *Suppl. Vol.* Proceedings *International Conference on Dependable Systems and Networks (DSN-2002),* (Whasington DC, USA), pp. F.7-F.8, 2002. - http://www.laas.fr/DBench.

[Kao *et al.* 1993] W.-L. Kao, R. K. Iyer et D. Tang, "FINE: A Fault Injection and Monitoring Environment for Tracing the UNIX System Behavior under Faults", *IEEE Transactions on Software Engineering*, 19 (11), pp.1105-1118, 1993.

[Koopman & DeVale 1999] P. Koopman et J. DeVale, "Comparing the Robustness of POSIX Operating Systems", in *Proceedings 29th International Symposium on Fault-Tolerant Computing (FTCS-29),* (Madison, WI, USA), pp.30-37, IEEE CS Press, 1999.

[Koopman *et al.* 1997] P. J. Koopman, J. Sung, C. Dingman, D. P. Siewiorek et T. Marz, "Comparing Operating Systems using Robustness Benchmarks", in *Proceedings 16th International Symposium on Reliable Distributed Systems (SRDS-16),* (Durham, NC, USA), pp.72-79, IEEE CS Press, 1997.

[Laprie *et al.* 1995] J.-C. Laprie, J. Arlat, J. P. Blanquart, A. Costes, Y. Crouzet, Y. Deswarte, J. C. Fabre, H. Guillermain, M. Kaâniche, K. Kanoun, C. Mazet, D. Powell, C. Rabéjac et P. Thévenod, *Guide de la sûreté de fonctionnement,* 324p., Cépaduès-Editions, Toulouse, France, 1995.

[Lee & Iyer 1992] I. Lee et R. K. Iyer, "Analysis of Software Halts in the Tandem Guardian Operating System", in Proceedings *3rd International Symposium on Reliability Engineering,* pp.227-236, 1992.

[LinuxHQ 2002] The Linux Headquarters, http://www.linuxhq.com, 2002.

[LynuxWorks] LynuxWorks, *"Embedded Linux and Embedded Real-Time Operating Systems: BlueCat and LynxOS by LynuxWorks"*.

[LWN Coverity 2004] "Coverty's kernel code quality study" *on The Linux Weekly News*, http://lwn.net/Articles/115530/

[Madeira *et al.* 2002a] H. Madeira, J. Arlat, K. Buchacker, D. Costa, Y. Crouzet, M. Dal Cin, J. Durães, P. Gil, T. Jarboui, A. Johanson, K. Kanoun, L. Lemus, R. Lindström, J. J. Serrano, N. Suri et M. Vieira, *Dependability Benchmark Definition: DBench Prototypes*, Dependability Benchmarking Project (IST-2000-25425), 2002.

[Madeira *et al.* 2000] H. Madeira, D. Costa et M. Vieira, "On the Emulation of Software Faults by Software Fault Injection", in *Proceedings International Conference on Dependable Systems and Networks (DSN-2000),* (New York, NY, USA), pp.417-426, IEEE CS Press, 2000.

[Madeira *et al.* 2002b] H. Madeira, R. Some, F. Moreira, D. Costa et D. Rennels, "Experimental Evaluation of a COTS system for space applications", in Proceedings *International Conference on Dependable Systems and Networks (DSN-2002),* (Whasington DC, USA), pp.325-330, 2002.

[Manuel Intel] *IA-32 Intel Architecture Software Developer's Manual*, volume 1 : *Basic Architecture*, Order Number 245470, volume 2 : *Instruction Set Reference*, Order Number 245471, volume 3 : *System Programming Guide*, Order Number 245472

[Marsden & Fabre 2001] E. Marsden et J.-C. Fabre, "Failure Mode Analysis of CORBA Service Implementations", in *Proceedings IFIP/ACM International Conference on Distributed Systems Platforms (Middleware'2001),* (Heidelberg, Germany), 2001.

[McVoy & Staelin 1996] L. W. McVoy et C. Staelin, "lmbench: Portable Tools for Performance Analysis", in Proceedings *USENIX Annuel Technical Conference,* (San Diego, CA, USA), pp.279-295, 1996.

[Microsoft DDK 2002] Microsoft, "DDK - Kernel Mode Driver Architecture", in http://www.micosoft.com/, 2002.

[Microsoft WDM 2002] Microsoft, "Introduction to the Windows Driver Model", in http://www.micosoft.com/hwdev/driver/wdm, 2002.

[Miller *et al.* 2000] B. P. Miller, D. Koski, C. P. Lee, V. Maganty, R. Murthy, A. Natarajan et J. Steidl, *Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services*, University of Wisconsin, USA, Research Report, février 2000.

[Mukherjee & Siewiorek 1997] A. Mukherjee et D. P. Siewiorek, "Measuring Software Dependability by Robustness Benchmarking", *IEEE Transactions of Software Engineering*, 23 (6) 1997.

[Murphy & Levidow 2000] B. Murphy et B. Levidow, "Windows 2000 Dependability", in Proceedings *Workshop on Dependable Networks and Operating Systems, at the International Conference on Dependable Systems and Networks (DSN-2000),* (New York, NY, USA), pp.D20-D28, 2000.

[Nutt 2000] G. Nutt, *Operating Systems: A modern Perspective,* 2000.

[Oney 1999] W. Oney, *Programming the Microsoft Windows Driver Model*, Microsoft, 1999.

[Ousterhout 1990] J. Ousterhout, "Why aren't Operating Systems Getting Faster as Fast as Hardware", in *Proceedings of the 1990 Summer USENIX Technical Conference,* (Anaheim, CA, USA), pp.247-256, 1990.

[Pan *et al.* 2001] J. Pan, P. J. Koopman, D. P. Siewiorek, Y. Huang, R. Gruber et M. L. Jiang, "Robustness Testing and Hardening of CORBA ORB Implementations", in *Proceedings International Conference on Dependable Systems and Networks (DSN-2001),* (Göteborg, Sweden), pp.141-150, IEEE CS Press, 2001.

[Rabah & Kanoun 1999] M. Rabah et K. Kanoun, "Dependability Evaluation of a Distributed Shared Memory Multiprocessor System", in Proceedings *3rd European Dependable Computing Conference (EDCC-3),* (A. Pataricza, J. Hlavicka and E. Maehle, Eds.), (Prague, Czech Republic), pp.42-59, Springer, 1999.

[Réveillère & Muller 2001] L. Réveillère, G. Muller, "Improving Driver Robustness: An Evaluation of the Devil Approach," in Proceedings *International Conference on Dependable Systems and Networks (DSN 2002)*, Göteborg, Sweden, 2001, pp. 131-140.

[Rimén *et al.* 1994] M. Rimén, J. Ohlsson et J. Torin, "On Microprocessor Error Behavior Modeling", in Proceedings *24th International Symposium on Fault Toulerant Computing (FTCS-24),* (Austin, Texas, USA), pp.76-85, 1994.

[Rodriguez *et al.* 2002] C. Rodriguez, A. Albinet et J. Arlat, "MAFALDA-RT: A Tool for Dependability Assessment of Real-Time Systems", in Proceedings *International Conference on Dependable Systems and Networks (DSN 2002),* (Washington, D.C., USA), pp.267-272, 2002.

[Rodríguez *et al.* 1999] M. Rodríguez, F. Salles, J.-C. Fabre et J. Arlat, "MAFALDA: Microkernel Assessment by Fault Injection and Design Aid", in *Proceedings 3rd European*

Dependable Computing Conference (EDCC-3), (E. M. J. Hlavicka, A. Pataricza, Ed.), (Prague, Czech Republic), LNCS, 1667, pp.143-160, Springer, 1999.

[Rubini & Corbet 2001] A. Rubini et J. Corbet, *Linux Device drivers,* O'Reilly (ISBN : 0-596-00008-1), 2001.

[Russinovich 99] M. Russinovich, Inside Win2K Reliablity Enhancements, Part 1,2 and 3, http://www.winntmag.com/Articles/Index.cfm?ArticleID=5766, 1999.

[Salles *et al*. 1999] F. Salles, M. Rodríguez, J.-C. Fabre et J. Arlat, "Metakernels and Fault Containment Wrappers", in *Proceedings 29th IEEE International Symposium on Fault-Tolerant Computing (FTCS-29),* (Madison, WI, USA), pp.22-29, IEEE CS Press, 1999.

[Sanders *et al*. 1995] W. H. Sanders, W. D. Obal II, M. A. Qureshi et F. K. Widjanarko, "The *UltraSAN* Modeling Environment", *Performance Evaluation*, 21 (special "Performance Evaluation Tools") 1995.

[Seltzer *et al*. 1996] M. I. Seltzer, Y. Endo, C. Small, et K. A. Smith, "Dealing with disaster: Surviving misbehaved kernel extensions", in *Proceedings of the 2nd OSDI*, Seattle, whashington, pp 213-227, 1995.

[Shelton *et al*. 2000] C. Shelton, P. Koopman et K. D. Vale, "Robustness Testing of the Microsoft Win32 API", in *Proceedings International Conference on Dependable Systems and Networks (DSN'2000),* (New York, NY, USA), IEEE CS Press, 2000.

[Siewiorek *et al*. 1993] D. P. Siewiorek, J. J. Hudak, B.-H. Suh et Z. Segall, "Development of a Benchmark to Measure System Robustness", in *Proceedings 23rd International Symposium on Fault-Tolerant Computing (FTCS-23),* (Toulouse, France), pp.88-97, IEEE CS Press, 1993.

[Smith 1995] B. Smith, "BYTE's UNIX Benchmarks", Personal Communication, 1995.

[Solomon & Russinovich 2000] D. A. Solomon et M. E. Russinovich, *Inside Microsoft Windows 2000, Third Edition,* 2000.

[Swift *et al*. 2003] M. M. Swift, B. N. Bershad et H. M. Levy, "Improving the Reliability of Commodity Operating Systems", in *21th ACM Symposium on Operating Systems Principles (SOSP-2003),* ACM Press, 2003.

[Swift *et al*. 2004] M. M. Swift, M. Annamalai, B. N. Bershad et H. M. Levy, "Recovering Device Drivers", in *6th Symposium on Operating Systems Design and Implementation (OSDI'2004).*

[Systems 1997] C. Systems, *CHORUS/ClassiX release 3 - Technical Overview*, Chorus Systems, N°Rapport. CS/TR-96-119.12, 1997.

[Tang & Iyer 1992] D. Tang et R. K. Iyer, "Analysis of the vax/vms error logs in multicomputer environments - A case study of software dependability", in Proceedings *3rd International Symposium on Reliability Engineering,* pp.216-226, 1992.

[Tsai & Singh 2000] T. Tsai et N. Singh, "Reliability Testing of Applications on Windows NT", in *Proceedings International Conference on Dependable Systems and Networks (DSN-2000),* (New York, NY, USA), pp.427-436, IEEE CS Press, 2000.

[Tsai *et al*. 1996] T. K. Tsai, R. K. Iyer et D. Jewitt, “An Approach Towards Benchmarking of Fault-Tolerant Commercial Systems”, in *Proceedings 26th International Symposium on Fault-Tolerant Computing (FTCS-26),* (Sendai, Japan), pp.314-323, IEEE CS Press, 1996.

[Tsegaye & Foss 2004] M. Tsegaye et R. Foss,"Acomparaison of the Linux and Windows Device Driver Architectures", in *ACM Operating System Review,* 38(2) , pp.8-33, 2004.

[Yaghmour & Dagenais 2000] K. Yaghmour et M. R. Dagenais, “Measuring and Characterizing System Behavior Using Kernel-Level Event Logging”, in Proceedings *USENIX Annual Technical Conference,* (San Diego, CA, USA), 2000.

[Zaatar & Ouaiss 2002] W. Zaatar, I. Ouaiss, “A Comparative Study of Device Driver APIs Towards A Uniform Linux Approach,” Proc. Ottawa Linux Symp., Ottawa, ON, Canada, 2002, pp. 407-413.

Table des matières