FlexOS : Vers une Isolation Flexible du Noyau

Hugo Lefeuvre, Vlad-Andrei Bâdoiu, Pierre Olivier, Tiberiu Mosnoi, Râzvan Deaconescu, Felipe Huici, Costin Raiciu

Compas'2021 : Parallélisme/ Architecture/ Système / Temps Réel
MILC - Lyon, France, 5-9 juillet 2021

FlexOS : Vers une Isolation Flexible du Noyau

Hugo Lefeuvre[1], Vlad-Andrei Bădoiu[2], Ștefan Teodorescu[2], Pierre Olivier[1], Tiberiu Mosnoi[2], Răzvan Deaconescu[2], Felipe Huici[3], et Costin Raiciu[2]

[1] The University of Manchester, [2] University Politehnica of Bucharest, [3] NEC Laboratories Europe

Résumé

Au moment de leur conception, les systèmes d'exploitation modernes implémentent une stratégie de sécurité et d'isolation bien précise reposant sur un ou plusieurs mécanismes logiciels ou matériels. Pour des raisons de coût, ce choix est rarement revisité après déploiement. Cette approche classique est limitée lorsque les protections matérielles viennent à casser, lorsque le matériel devient hétérogène, ou lorsque l'on cherche à spécialiser dynamiquement le système pour un cas d'usage précis. Nous présentons *FlexOS*, un système d'exploitation permettant de facilement spécialiser la stratégie d'isolation du noyau à la compilation et non à la conception. FlexOS est un LibOS modulable incluant des composants isolables à des granularités variables via de multiples mécanismes, d'un langage de description permettant à l'utilisateur de détailler les besoins en sécurité du système, et d'un framework d'exploration automatique des compromis de sécurité et performance offerts par FlexOS pour une application donnée. Nous évaluons FlexOS et démontrons le vaste espace de conception qu'il permet d'explorer.

1 Introduction

Afin de créer des logiciels sécurisés, les développeurs disposent de trois principales approches offrant divers compromis quant au coût d'ingénierie, aux garanties de sécurité, ainsi qu'aux performances : la vérification formelle, le *software runtime checking*, et l'isolation matérielle. Les logiciels modernes reposent sur une ou plusieurs de ces approches, et ce choix définitif est réalisé tôt dans le flot de conception du logiciel : changer de modèle d'isolation après déploiement est coûteux et donc rare.

Sur le plan des systèmes d'exploitation (*Operating System*, OS), le paysage actuel, illustré sur la Figure 1, inclut : les micro-noyaux [20, 25], favorisant l'isolation matérielle (à gros grain, celui d'un sous-système) et la vérification par rapport à la performance ; les noyaux monolithiques [8], qui privilégient les performances et isolent les processus mais font confiance au noyau dans son ensemble ; et les OS à espace d'adressage unique (*single address-space OS*) qui isolent au sein de l'espace d'adressage [9, 28, 19], ou rejettent toute forme de compartimentation pour privilégier la performance pure [30, 35, 6, 26].

Les design de ces OS sont fortement liés aux mécanismes de protection qu'ils implémentent, rendant des changements sur ce plan particulièrement complexes à réaliser : retirer la séparation *user/kernel* d'un noyau monolithique [31] requiert un effort d'ingénierie important, tout comme le découpage d'un noyau monolithique en plusieurs entités distinctes [23]. Malgré cette rigidité, nombre de méthodes d'isolation sont capables d'offrir des garanties équivalentes, en termes d'isolation mémoire [16, 44], de contrôle de pre/post-conditions à l'exécution [39], du flux d'exécution [45], etc.

L'utilisation rigide de primitives de sécurité dans les OS modernes pose plusieurs problèmes. Premièrement, réagir rapidement et de manière efficace à des situations où les protections offertes par des mécanismes matériels cassent (*Meltdown, etc.*) est difficile et coûteux. Lorsque plusieurs mécanismes peuvent être utilisés pour la même tâche (par exemple SFI ou la vérification), choisir la primitive qui offre la meilleure performance dépend de nombreux facteurs tels que le matériel disponible, la charge de travail, etc. Ce choix devrait idéalement être repoussé au moment du déploiement et non à la conception, dans le but de pouvoir adapter aisément la stratégie de sécurité et les performance à divers cas d'usage applicatifs. Le matériel est également de plus en plus hétérogène [46] et certaines primitives d'isolation ne sont offertes que par un certain type de matériel ou processeur (*memory protection keys* [10] (MPK), *hardware capabilities* [47], etc.). Lorsque le même logiciel doit fonctionner sur différents types de matériel, *comment minimiser l'effort de portage tout en préservant la sécurité du système ?*

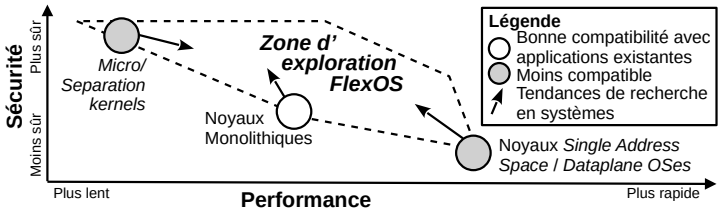

FIGURE 1 – Espace de conception des OS modernes.

Enfin, une stratégie d'isolation fixée à la conception rend difficile les approches de protection incrémentales : alors qu'il est extrêmement coûteux de vérifier un OS complet [25], il est courant de vérifier un composant/sous-système : se pose alors la question du maintien des garanties de sécurité lors de l'intégration d'un composant vérifié dans une base de code noyau non-vérifiée. Cela nous mène au problème de recherche suivant : *Comment permettre aux utilisateurs d'un OS de changer de stratégie et de mécanisme d'isolation de manière simple et sécuritaire, au moment du déploiement et non de la conception du système ?*
Notre réponse à ce problème est *FlexOS*, un noyau dont la conception modulaire permet d'adapter facilement et à moindre coût le profil d'isolation et de protection du système pour une application ou un cas d'usage particulier, et ce au moment de la compilation et non de la conception, comme c'est le cas avec les solutions actuelles. Dans ce but, nous nous basons sur le concept d'OS bibliothèque (*library OS*/libOS), en augmentant sa capacité historique de spécialisation pour la performance [15, 22] d'une capacité de spécialisation pour la *sécurité*. Avec FlexOS, un opérateur peut décider, au moment de la compilation et donc au déploiement, quels composants du système doivent être isolés et comment, et ce avec une granularité fine. FlexOS permet d'explorer facilement le vaste espace de conception qu'offrent la multitude de technologies d'isolation et leur différentes granularités, afin de choisir le meilleur profil sécurité/performance pour un cas d'usage applicatif spécifique. Concrètement, cet article propose les contributions suivantes :

— La conception et l'implémentation d'un prototype de FlexOS, un noyau et ensemble d'outils pour explorer facilement les compromis sécurité/performance des OS modernes ;
— L'identification des primitives fondamentales nécessaires pour offrir des garanties d'isolation reposant sur un vaste ensemble de mécanismes logiciels et matériels ;
— Une évaluation préliminaire montrant que FlexOS peut être utilisé pour explorer un vaste ensemble de profils sécurité/performance sur deux applications : iperf et Redis.

2 Design et Implémentation d'un Prototype

Le but de FlexOS est de permettre aux développeurs et chercheurs en systèmes d'explorer facilement les nombreux compromis de sécurité et de performance de l'isolation du noyau. Explorer un tel espace n'est pas une tâche évidente, du fait du large nombre de configurations possibles, et de la difficulté de quantifier performances et sécurité. Pour cela plusieurs stratégies peuvent être suivies, notamment :

— Pour un budget de performance donné (par exemple, une méthode de mesure de la performance et une valeur de référence) et un ensemble de compartiments prédéfini (par exemple, isoler l'application et la pile réseau du reste du noyau), on souhaite trouver la stratégie de compartimentation et les mécanismes qui maximisent la sécurité ;
— Pour un but de sécurité donné (par exemple, pas de dépassements de tampons), on souhaite trouver une conformation qui offre la meilleure performance ou fonctionne sur le plus grand nombre de machines (basé, par exemple, sur une liste de machines donnée).

Ces deux objectifs ont en commun la nécessité de décrire le degré de sécurité associé à chaque mécanisme, ainsi que les implications relatives à l'exécution de deux composants logiciels donnés dans un même compartiment.
Afin de répondre à ces besoins, nous basons le design de FlexOS sur le modèle LibOS [15], profitant ainsi de sa modularité à grain fin. Notre approche est de supporter un ensemble de mécanismes logiciels et matériels, et de complémenter l'API de chaque module noyau avec des *métadonnées FlexOS* spécifiant : (1) le comportement attendu vis à vis d'autres composants sur le plan de l'accès mémoire (par exemple, afin que les propriétés du module soient préservées) ; (2) les zones de mémoire auxquelles le module peut accéder à la fois dans un mode d'opération usuel mais aussi antagoniste (par exemple dans le

cas où son flux de contrôle serait compromis) ; et (3) des informations sur l'API du module. De telles métadonnées sont saisies pour chaque module noyau par leur développeurs. Cela représente un effort relativement faible et surtout unique dans la vie du module. Ces métadonnées FlexOS permettent à un système automatisé de raisonner sur les conséquences en terme de sécurité du choix d'exécuter deux modules donnés dans un même compartiment. Par exemple, voici un extrait des métadonnées décrivant les propriétés d'un ordonnanceur formellement vérifié que nous avons implémenté en Dafny [27] :

```
[Memory access] Read(Own,Shared); Write(Own,Shared)
[Call] alloc::malloc, alloc::free
[API] thread_add (...); thread_remove(...); yield(...)
[Requires] *(Read,Own), *(Write,Shared), *(Call, thread_add), *...
```

Cette description spécifie que (1) l'ordonnanceur n'accède qu'à sa propre mémoire et à un segment partagé avec d'autres bibliothèques (par exemple, celles appelant les fonctions de son API) ; (2) qu'il n'utilise que les fonctions exposées par l'allocateur mémoire ; (3) quelles fonctions la bibliothèque expose via son API ; (4) que l'ordonnanceur attend que d'autres bibliothèques soient capable de lire sa mémoire (mais pas d'y écrire) et d'être capable de lire et écrire dans la mémoire partagée. Au delà de composants logiciels vérifiés, cette description sera relativement similaire dans le cas de composants écrits dans des langages *memory-safe*.

Considérons maintenant un composant écrit dans un langage *memory-unsafe* comme le C (*e.g.,* une pile TCP/IP). Ce composant sera considéré comme potentiellement dangereux ; on considérera que son *control/data flow* peut être détourné au cours de l'exécution (potentiellement résultant en des accès mémoire arbitraires sur l'intégralité de l'espace d'adressage), et il n'y aura pas de clause `Requires` puisqu'il n'y a pas de préconditions particulières à satisfaire. Sa description sera donc la suivante :

```
[Memory access] Read(*); Write(*).
[Call] *
```

Étant donné deux bibliothèques et leur métadonnées FlexOS, nous disposons maintenant d'assez d'informations pour décider automatiquement si ces dernières peuvent fonctionner dans le même compartiment sans perte de garanties. Si elles ne disposent pas de clause `Requires`, la réponse est oui. Si l'une des deux prévoit une telle clause, il est possible de vérifier automatiquement sa capacité a s'exécuter dans le même domaine de sécurité que l'autre bibliothèque. Prenons l'exemple ci-dessus. Afin de valider ses préconditions, l'ordonnanceur vérifié requiert que d'autres composants ne puissent que lire sa mémoire mais pas l'écrire. Le composant en C, d'un autre côté, peut écrire toute la mémoire à laquelle il a accès (dans le compartiment) — ainsi les attentes de l'ordonnanceur vérifié ne sont pas satisfaites et les deux modules ne peuvent être disposés dans un même compartiment sans pertes de garanties.

Armé avec ces informations de compatibilité par paire, sélectionner le plus petit nombre de compartiments dans une image FlexOS peut être réduit au problème classique de coloration de graphe : chaque bibliothèque est un sommet, et une arrête relie deux bibliothèques incompatibles. La coloration de graphe associe le plus petit nombre de couleurs possible aux sommets de telle sorte qu'aucun sommet adjacent n'ait la même couleur. Nous instancions un compartiment par couleur. Dans le pire des cas, chaque bibliothèque reçoit son propre compartiment.

Quand activer la SFI ? Pour des raisons de performances ou de compatibilité il est parfois préférable d'utiliser des contrôles logiciels à l'exécution (CFI, DFI, etc., regroupés sous le terme *SFI*) à la place de plusieurs compartiments, et ce possiblement pour un sous-ensemble des modules du système. Afin de pouvoir automatiser le processus de sélection de mécanismes SFI, nous créons tout d'abord une description lisible à la machine de l'impact de chaque mécanisme sur les propriétés de sûreté d'une bibliothèque. Cette transformation prend en entrée les métadonnées FlexOS d'une bibliothèque et produit un nouvel ensemble de métadonnées décrivant les nouvelles propriétés de sûreté de la bibliothèque lorsque le mécanisme SFI est activé. Pour CFI, par exemple, la transformation est simple : les bibliothèques qui déclarent `Call(*)` sont transformées en `Call(liste de fonctions);` la liste de fonctions étant automatiquement déterminée par une analyse de flux de contrôle de la bibliothèque. Pour DFI, la transformation est similaire : si le graphe de flux de données montre que tout ce que la bibliothèque écrit est ses propres données, alors `Writes(*)` sera transformé en `Writes(Own);` d'autres techniques de SFI sont gérées de manières similaire. Pour énumérer tous les déploiements

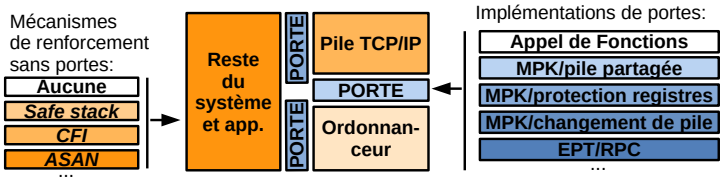

FIGURE 2 – Architecture de FlexOS.

possibles avec SFI, nous procédons comme suit : pour chaque bibliothèque qui écrit sur toute la mémoire, activer DFI/ASAN ; pour chaque bibliothèque qui peut exécuter du code arbitraire, activer CFI. Le résultat de cette étape est une liste de modules disponibles en deux versions ; avec et sans SFI. Nous parcourons ensuite toutes les combinaisons et exécutons l'algorithme de coloration de graphes discuté précédemment. Le résultat est autant de colorations possibles que de combinaisons de bibliothèques.

Gestion des Pré- et Postconditions. L'approche que nous avons pris précédemment peut aussi être appliquée aux préconditions que certains modules formellement vérifiés peuvent exiger avant l'appel de leurs fonctions. Dans ce cas, FlexOS peut automatiquement vérifier que les préconditions sont satisfaites lors d'un appel (basé sur une analyse statique du *call graph*) ; si elles ne sont pas satisfaites, alors des contrôles logiciels à l'exécution doivent êtres ajoutés pour garantir leur satisfaction. Dans notre prototype nous les ajoutons manuellement, mais nous souhaitons explorer des manières d'automatiser ce problème dans nos travaux futurs.

Architecture de FlexOS. FlexOS est basé sur Unikraft [26], un libOS modulaire. La modularité d'Unikraft rend l'isolation à grain fin de composants plus simple qu'avec un noyau monolithique classique, tout en présentant des performances à l'état de l'art. En reposant sur l'isolation de composants, FlexOS permet d'isoler des modules de manière flexible avec une granularité bien plus fine que dans un micro-noyau (voir Figure 2). Les compartiments de FlexOS communiquent par des *portes* qui représentent l'API exposée par une bibliothèque au reste du noyau. Les portes sont le médium de communication entre les bibliothèques. Par exemple les portes entre deux bibliothèques dans un même compartiment seront implémentées à l'aide d'appel de fonctions classiques ; pour une isolation avec Intel MPK il s'agira d'un appel de fonction couplé à un changement de clé MPK, etc. À noter que certains mécanismes ne requièrent pas de portes particulières ; dans le cas de la SFI par exemple, des appels de fonctions classiques sont suffisants. Les portes sont ainsi instanciées au moment de la compilation en fonction d'une description de la conformation du noyau (fournie par l'utilisateur ou par un outil automatisé). L'utilisation de telles portes nécessite le remplacement dans le code source de tous les appels de fonctions entre bibliothèques par des portes temporaires, qui seront elles-mêmes remplacées par une implémentation spécifique à la compilation. Pour notre prototype ce travail est réalisé manuellement, bien qu'il puisse être automatisé. En plus de ces annotations du *code*, le portage requiert également des annotations des *données* qui doivent être partagées entre deux bibliothèques communiquantes. En utilisant ces primitives de base, un développeur peut facilement développer une variété de stratégies et de mécanismes d'isolation pour donner à l'utilisateur la possibilité de les explorer et de trouver l'implémentation la plus rapide ou la plus sûre pour un cas donné.

Implémentation d'un Prototype. Afin d'illustrer la faisabilité de notre concept, nous avons implémenté un prototype de FlexOS sur la base d'Unikraft v0.4 [43] (1.5K lignes de code). Notre prototype prend en charge trois primitives d'isolation : Intel MPK, Machine Virtuelle (EPT), et SFI.
Intel MPK est un mécanisme d'isolation mémoire au sein d'un espace d'adressage [1, 5, 38]. Notre prototype place chaque compartiment dans sa propre zone MPK, incluant la mémoire statique, le tas, et la pile. Le support MPK est concrétisé par deux types de portes inter-compartiments, permettant ou non de partager la pile, chaque option offrant un rapport sécurité/performance différent.
Concernant l'isolation via l'hyperviseur, EPT offre une compatibilité et un niveau d'isolation supérieurs à MPK, au prix d'une dégradation de performances venant de l'usage de multiples espaces d'adressage : la chaîne d'outils de FlexOS génère autant d'images de VM que de compartiments. Chaque image contient le code plate-forme minimum nécessaire pour fonctionner indépendemment ainsi que les bibliothèques qu'elle abrite et une implémentation de *remote procedure calls* légère sur la base de

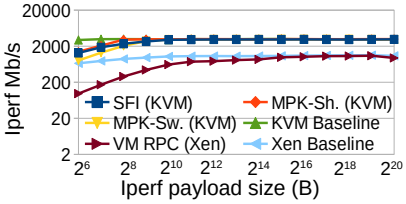

FIGURE 3 – débit iperf, différentes configurations (*Sh* - pile d'exécution partagée, *Sw* - isolées)

Composant *C*	**SFI: tout sauf *C***	**SFI: *C seulement***
Ordonnanceur	496Mb/s	2.90 Gb/s
Pile TCP/IP	631Mb/s	2.76 Gb/s
LibC	1.47Gb/s	1.25 Gb/s
Reste du système	1.08Gb/s	2.50 Gb/s
Système entier	2.94Gb/s (*référentiel*)	489 Mb/s

TABLE 1 – débit iperf avec SFI sur divers composants.

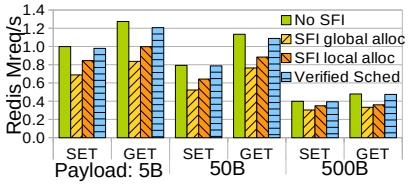

FIGURE 4 – Débit Redis avec différentes configurations de SFI et notre ordonnanceur vérifié.

mémoire partagée et de notifications inter-VM pour la communication entre compartiments. Notre support s'appuie sur l'hyperviseur Xen avec une implémentation pour KVM en cours.
Concernant SFI, FlexOS prend en charge *KASAN* [41], *stack protector* [14], *UBSAN* [42], *CFI* [2, 3] et *SafeStack* [40], de manière modulaire : nous pouvons appliquer les différents mécanismes à la granularité d'une bibliothèque et non du système, pour une protection à grain fin et des compromis sécurité/performance flexibles. Cela est rendu possible par la capacité de FlexOS à instancier un allocateur de mémoire par compartiment : de nombreuses techniques de SFI requièrent un allocateur mémoire instrumenté moins performant, et notre design modulaire permet d'éviter de payer ce prix pour l'intégralité du système.
Nous avons implémenté un ordonnanceur vérifié en Dafny [27], qui génère du code C++ qui s'intègre aisément à FlexOS. Notre ordonnanceur offre des garanties prouvées statiquement et validées par des pré-/postconditions imposées dynamiquement dans le code d'interfaçage avec le reste du système.

3 Résultats Préliminaires

Nous souhaitons confirmer que FlexOS permet une exploration facile des compromis de sécurité/performance. Nous étudions les performances de plusieurs configurations FlexOS pour deux applications : un serveur iperf et Redis. Chaque configuration est obtenue en modifiant quelques options et en recompilant. Les mesures sont réalisées sur un processeur Xeon 4110 (2.1 GHz).

iperf : Isolation de la Pile TCP/IP. Nous instancions un serveur iperf dont la pile réseau est isolée du reste du système, dans 3 configurations : (1) deux compartiments avec MPK — un pour la pile TCP/IP et un pour le reste du système ; (2) deux VMs isolant la pile TCP/IP du reste du système ; (3) isolation de la pile TCP/IP via SFI. Nous varions la taille du tampon passé à `recv` pour capturer des effets de *batching*. Les résultats de performance sont visibles sur la Figure 3. Avec SFI et MPK, le coût en terme de performance est élevé pour de petits tampons (2-3x). Avec de plus grandes tailles ces solutions se rapprochent rapidement du référentiel (performance similaire après 1KB). Le référentiel Xen est inférieur à KVM dû à un manque d'optimisation d'Unikraft ; on observe cependant que l'isolation par VM nécessite un effet de *batching* plus important pour rattraper le référentiel (32KB), dû aux coûts de changement de contexte plus élevés. Ces résultats montrent que l'impact en terme de performance de différentes solutions dépend de la charge de travail. Les résultats varient fortement en fonction des mécanismes, et le choix d'une solution en particulier n'est pas optimal de manière générale.

SFI à Grain Fin & Vérification. Le design modulaire de FlexOS nous permet d'activer SFI à la granularité d'une bibliothèque. Afin d'illustrer cette capacité, nous avons d'abord instancié iperf en activant SFI pour différents composants : la pile TCP/IP, l'ordonnanceur, la libC, et le reste du système (iperf inclus). Les résultats sont visibles sur la Table 1. L'impact en performance varie en fonction du composant pour lequel SFI est activé : le ralentissement est presque nul pour l'ordonnanceur (1%) mais explose pour la libC (x2.3). Le coût pour la pile TCP/IP est assez faible (6%). SFI pour le système entier implique un ralentissement de x6, montrant les bénéfices de la flexibilité de FlexOS.
Une seconde expérience étudie plus en détail SFI pour la pile TCP/IP. Nous avons instancié Redis avec différentes configurations de SFI : (1) un référentiel sans SFI ; (2) un allocateur instrumenté pour l'intégralité du système ; et (3) un allocateur instrumenté dédié pour la pile TCP/IP. Les résultats sont visibles sur la Figure 4. Avec l'allocateur global, le ralentissement pour le système entier est de 1.45x. La capacité de FlexOS à instancier un allocateur par composant permet de réduire ce coût à 1.25x.
Enfin, nous avons évalué les compromis que peut offrir un composant vérifié comme l'ordonnanceur

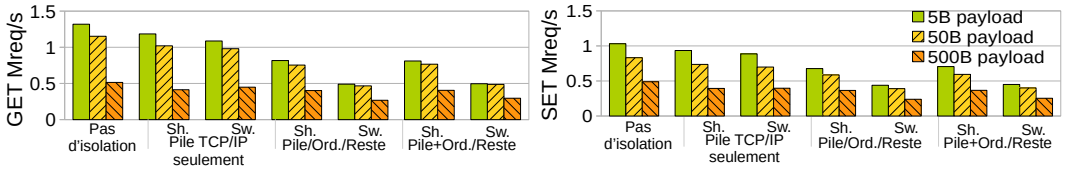

FIGURE 5 – Débit Redis pour différents scénarios d'isolation MPK.

décrit précédemment. Nos micro-benchmarks montrent un coût moyen de changement de contexte en moyenne trois fois plus élevé que pour l'ordonnanceur C : 218.6ns contre 76.6ns, mais la Figure 4 montre que ce coût est raisonnable en pratique avec un ralentissement moyen de 6% pour Redis. L'utilisation flexible de SFI et de la vérification sur la base d'une bibliothèque permet de profiter de certains avantages de ces mécanismes sans en payer le coût, ouvrant la voie à de nombreux potentiels de couplage avec des mécanismes matériels.

Redis : Variation des Stratégies d'Isolation avec MPK. Nous avons évalué Redis dans une variété de scénarios sous Intel MPK. Nous avons défini quatre modèles d'isolation : *{ pile TCP/IP, reste du système }*, *{ pile TCP/IP, ordonnanceur, reste du système }*, et *{ pile TCP/IP + ordonnanceur, reste du système }*, ainsi qu'un référentiel sans isolation. Ces différents scénarios illustrent la capacité de FlexOS à simplement composer avec une variété de modèles de confiance. Pour chaque scénario nous avons réalisé les mesures avec et sans isolation de la pile d'exécution.

Les résultats sont visibles sur la Figure 5. Le coût de l'isolation varie en fonction du nombre de compartiments, de la porte utilisée, et de l'approche de communication empruntée par les compartiments. Isoler la pile réseau seulement entraîne un ralentissement de 17%. Isoler également l'ordonnanceur entraîne un ralentissement de 1.4x-2.25x en fonction de la porte utilisée. Ces résultats illustrent l'importance de la communication entre l'ordonnanceur et la pile réseau. Néanmoins, mettre la pile réseau et l'ordonnanceur dans le même compartiment n'améliore pas significativement la performance. La communication entre la pile réseau et l'ordonnanceur passe par la libC, ne réduisant ainsi pas le nombre de changements de contexte. En somme, ces résultats montrent la variété de compromis sécurité/performance qui peuvent être obtenus, mais aussi la complexité de raisonner sur la performance d'une configuration ; cela motive notre approche d'exploration automatique de l'espace de conception.

4 Travaux Connexes

Des précédents travaux ont cherché à améliorer la sécurité de noyaux monolithiques classiques en réduisant la taille de leur base de code de confiance par la séparation [36, 4], en se tournant vers des micro-noyaux [17, 20], et plus récemment en mettant en oeuvre des langages sûrs [7, 32, 21, 30, 11]. En ce qui concerne les OS à espace d'adressage unique/LibOS, de l'isolation peut être introduite via la table des pages [9, 28, 19], et plus récemment MPK [37, 29, 39, 34]. D'autres études se concentrent sur la vérification formelle [24, 25] et certains mécanismes de SFI légers [12, 14]. Les mécanismes logiciels les plus sûrs [13], néanmoins, sont rarement utilisés en production à cause de leur coût en performance. Chacune de ces approches représente un seul point dans l'espace de conception des OS. Peu d'études ont étudié des systèmes permettant une approche flexible vis à vis de l'isolation, à l'exception de LibrettOS [33] qui permet d'alterner entre micro-noyau et libOS, mais reste globalement limité à une faible portion de l'espace de conception. Enfin, SOAAP [18] propose un système pour explorer les compromis sécurité/performance en compartimentation, mais ce travail cible des bases de code *monolithiques et utilisateur*, alors que FlexOS se concentre sur des *noyaux d'OS modulaires*.

5 Conclusions

FlexOS permet à des opérateurs/chercheurs d'explorer aisément les compromis sécurité/performance offerts par différentes stratégies et mécanismes d'isolation dans le noyau, avec pour but de créer des systèmes sur mesure pour une application/un cas d'usage donné. Dans cet article, nous présentons un aperçu initial du potentiel de FlexOS.

Bibliographie

1. Intel® 64 and IA-32 Architectures Software Developer's Manual. Volume 3A, Section 4.6.2.

2. Abadi (M.), Budiu (M.), Erlingsson (U.) et Ligatti (J.). – Control-flow integrity. – In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, *CCS '05*, CCS '05, pp. 340–353, New York, NY, USA, 2005. Association for Computing Machinery.
3. Abadi (M.), Budiu (M.), Erlingsson (U.) et Ligatti (J.). – Control-flow integrity principles, implementations, and applications. *ACM Trans. Inf. Syst. Secur.*, vol. 13, n1, novembre 2009.
4. Alves-Foss (J.), Oman (P.), Taylor (C.) et Harrison (S.). – The mils architecture for high-assurance embedded systems. *Int. J. Embed. Syst.*, vol. 2, 2006, pp. 239–247.
5. Bannister (S.). – Memory tagging extension : Enhancing memory safety through architecture. – `https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/enhancing-memory-safety`, août 2019. Online ; accessed October 27, 2020.
6. Belay (A.), Prekas (G.), Klimovic (A.), Grossman (S.), Kozyrakis (C.) et Bugnion (E.). – IX : A protected dataplane operating system for high throughput and low latency. – In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*, *OSDI '14*, OSDI '14, pp. 49–65, Broomfield, CO, octobre 2014. USENIX Association.
7. Boos (K.), Liyanage (N.), Ijaz (R.) et Zhong (L.). – Theseus : an experiment in operating system structure and state management. – In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation*, *OSDI'20*, OSDI'20, pp. 1–19. USENIX Association, novembre 2020.
8. Bovet (D. P.) et Cesati (M.). – *Understanding the Linux Kernel : from I/O ports to process management*. – O'Reilly Media, Inc., 2005.
9. Chase (J. S.), Levy (H. M.), Feeley (M. J.) et Lazowska (E. D.). – Sharing and protection in a single-address-space operating system. *ACM Trans. Comput. Syst.*, vol. 12, n4, novembre 1994, p. 271–307.
10. Corbet (J.). – Memory protection keys. *Linux Weekly News*, 2015. – `https://lwn.net/Articles/643797/`.
11. Cutler (C.), Kaashoek (M. F.) et Morris (R. T.). – The benefits and costs of writing a POSIX kernel in a high-level language. – In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18)*, pp. 89–105, 2018.
12. Edge (J.). – Kernel address space layout randomization. – `https://lwn.net/Articles/569635/`, 2013.
13. Edge (J.). – The kernel address sanitizer. – `https://lwn.net/Articles/612153/`, 2014.
14. Edge (J.). – "strong" stack protection for GCC. – `https://lwn.net/Articles/584225/`, 2014.
15. Engler (D. R.), Kaashoek (M. F.) et O'Toole (J.). – Exokernel : An operating system architecture for application-level resource management. – In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, *SOSP '95*, SOSP '95, p. 251–266, New York, NY, USA, 1995. Association for Computing Machinery.
16. Fleming (M.). – A thorough introduction to eBPF. – `https://lwn.net/Articles/740157/`, 2017.
17. Golub (D. B.), Julin (D. P.), Rashid (R. F.), Draves (R. P.), Dean (R. W.), Forin (A.), Barrera (J.), Tokuda (H.), Malan (G.) et Bohman (D.). – Microkernel operating system architecture and mach. – In *In Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pp. 11–30, 1992.
18. Gudka (K.), Watson (R. N.), Anderson (J.), Chisnall (D.), Davis (B.), Laurie (B.), Marinos (I.), Neumann (P. G.) et Richardson (A.). – Clean application compartmentalization with soaap. – In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, *CCS '15*, CCS '15, pp. 1016–1031, New York, NY, USA, 2015. Association for Computing Machinery.
19. Heiser (G.), Elphinstone (K.), Vochteloo (J.), Russell (S.) et Liedtke (J.). – The mungi single-address-space operating system. *Software : Practice and Experience*, vol. 28, n9, juillet 1999, p. 901–928.
20. Herder (J. N.), Bos (H.), Gras (B.), Homburg (P.) et Tanenbaum (A. S.). – Minix 3 : A highly reliable, self-repairing operating system. *SIGOPS Oper. Syst. Rev.*, vol. 40, n3, juillet 2006, p. 80–89.
21. Hunt (G. C.) et Larus (J. R.). – Singularity : Rethinking the software stack. *SIGOPS Oper. Syst. Rev.*, vol. 41, n2, avril 2007, pp. 37–49.
22. Kaashoek (M. F.), Engler (D. R.), Ganger (G. R.), Briceno (H. M.), Hunt (R.), Mazieres (D.), Pinckney

(T.), Grimm (R.), Jannotti (J.) et Mackenzie (K.). – Application performance and flexibility on exokernel systems. – In *Proceedings of the sixteenth ACM symposium on Operating systems principles*, pp. 52–65, 1997.

23. Kilpatrick (D.). – Privman : A library for partitioning applications. – In *USENIX Annual Technical Conference, FREENIX Track*, pp. 273–284, 2003.
24. Klein (G.). – Operating system verification—an overview. *Sadhana*, vol. 34, n1, 2009, pp. 27–69.
25. Klein (G.), Elphinstone (K.), Heiser (G.), Andronick (J.), Cock (D.), Derrin (P.), Elkaduwe (D.), Engelhardt (K.), Kolanski (R.), Norrish (M.), Sewell (T.), Tuch (H.) et Winwood (S.). – Sel4 : Formal verification of an os kernel. – In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, *SOSP '09*, SOSP '09, p. 207–220, New York, NY, USA, 2009. Association for Computing Machinery.
26. Kuenzer (S.), Bădoiu (V.-A.), Lefeuvre (H.), Santhanam (S.), Jung (A.), Gain (G.), Soldani (C.), Lupu (C.), Teodorescu (c.), Răducanu (C.), Banu (C.), Mathy (L.), Deaconescu (R.), Raiciu (C.) et Huici (F.). – Unikraft : Fast, specialized unikernels the easy way. – In *Proceedings of the 16th European Conference on Computer Systems*, *EuroSys '21*, EuroSys '21, pp. 376–394, New York, NY, USA, 2021. Association for Computing Machinery.
27. Leino (K. R. M.). – Dafny : An automatic program verifier for functional correctness. – In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, *LPAR'10*, LPAR'10, p. 348–370, Berlin, Heidelberg, 2010. Springer-Verlag.
28. Leslie (I. M.), McAuley (D.), Black (R.), Roscoe (T.), Barham (P.), Evers (D.), Fairbairns (R.) et Hyden (E.). – The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal on Selected Areas in Communications*, vol. 14, n7, 1996, pp. 1280–1297.
29. Li (G.), Du (D.) et Xia (Y.). – Iso-UniK : lightweight multi-process unikernel through memory protection keys. *Cybersecurity*, vol. 3, n1, mai 2020, p. 11.
30. Madhavapeddy (A.), Mortier (R.), Rotsos (C.), Scott (D.), Singh (B.), Gazagnaire (T.), Smith (S.), Hand (S.) et Crowcroft (J.). – Unikernels : Library operating systems for the cloud. – In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, *ASPLOS '13*, ASPLOS '13, pp. 461–472. Association for Computing Machinery, 2013.
31. Maeda (T.) et Yonezawa (A.). – Kernel mode linux : Toward an operating system protected by a type theory. – In *Annual Asian Computing Science Conference*, pp. 3–17. Springer, 2003.
32. Narayanan (V.), Huang (T.), Detweiler (D.), Appel (D.), Li (Z.), Zellweger (G.) et Burtsev (A.). – Redleaf : Isolation and communication in a safe operating system. – In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation*, *OSDI'20*, OSDI'20. USENIX Association, novembre 2020.
33. Nikolaev (R.), Sung (M.) et Ravindran (B.). – Librettos : A dynamically adaptable multiserver-library os. – In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, *VEE '20*, VEE '20, p. 114–128, New York, NY, USA, 2020. Association for Computing Machinery.
34. Olivier (P.), Barbalace (A.) et Ravindran (B.). – The case for intra-unikernel isolation. *Proceedings of the 10th Workshop on Systems for Post-Moore Architectures*, avril 2020.
35. Peter (S.), Li (J.), Zhang (I.), Ports (D. R.), Woos (D.), Krishnamurthy (A.), Anderson (T.) et Roscoe (T.). – Arrakis : The operating system is the control plane. *ACM Transactions on Computer Systems (TOCS)*, vol. 33, n4, 2015, pp. 1–30.
36. Rushby (J. M.). – Design and verification of secure systems. – In *Proceedings of the 8th ACM Symposium on Operating Systems Principles*, *SOSP '81*, SOSP '81, p. 12–21, New York, NY, USA, 1981. Association for Computing Machinery.
37. Sartakov (V. A.), Vilanova (L.) et Pietzuch (P.). – CubicleOS : A library OS with software componentisation for practical isolation (extended abstract). – In *Proceedings of the 26th International Conference on Architectural Support for Programming Languages and Operating Systems*, *ASPLOS*, ASPLOS, 2021.
38. Schrammel (D.), Weiser (S.), Steinegger (S.), Schwarzl (M.), Schwarz (M.), Mangard (S.) et Gruss (D.). – Donky : Domain keys – efficient in-process isolation for risc-v and x86. – In *Proceedings of the 29th USENIX Security Symposium*, *USENIX Security'20*, USENIX Security'20, pp. 1677–1694. USENIX Association, août 2020.

39. Sung (M.), Olivier (P.), Lankes (S.) et Ravindran (B.). – Intra-unikernel isolation with intel memory protection keys. – In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, *VEE '20*, VEE '20, p. 143–156, New York, NY, USA, 2020. Association for Computing Machinery.
40. The Clang Development Community. – Clang 13 documentation – safestack. – `https://clang.llvm.org/docs/SafeStack.html`, 2021. Online ; accessed Jul, 01 2021.
41. The Linux Kernel Development Community. – The kernel address sanitizer (KASAN). – `https://www.kernel.org/doc/html/v5.10/dev-tools/kasan.html`, 2020. Online ; accessed Jan, 25 2021.
42. The Linux Kernel Development Community. – The undefined behavior sanitizer (UBSAN). – `https://www.kernel.org/doc/html/v5.10/dev-tools/ubsan.html`, 2020. Online ; accessed Jan, 25 2021.
43. Unikraft Contributors. – Unikraft release 0.4. – `https://github.com/unikraft/unikraft/tree/RELEASE-0.4`, 2020.
44. Wahbe (R.), Lucco (S.), Anderson (T. E.) et Graham (S. L.). – Efficient software-based fault isolation. – In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, *SOSP '93*, SOSP '93, p. 203–216, New York, NY, USA, 1993. Association for Computing Machinery.
45. Wang (J.). – initial control flow support for ebpf verifier. – `https://lwn.net/Articles/753724/`, 2018.
46. Wang (X.), Yeoh (S.), Lyerly (R.), Olivier (P.), Kim (S.-H.) et Ravindran (B.). – A framework for software diversification with isa heterogeneity. – In *23rd International Symposium on Research in Attacks, Intrusions and Defenses ({RAID} 2020)*, pp. 427–442, 2020.
47. Watson (R. N.), Woodruff (J.), Neumann (P. G.), Moore (S. W.), Anderson (J.), Chisnall (D.), Dave (N.), Davis (B.), Gudka (K.), Laurie (B.) et al. – CHERI : A hybrid capability-system architecture for scalable software compartmentalization. – In *2015 IEEE Symposium on Security and Privacy*, pp. 20–37. IEEE, 2015.