Preuve formelle d'isolation mémoire dynamique à base de MMU

Narjes Jomaa, David Nowak, Gilles Grimaud, Julien Iguchi-Cartigny

Preuve formelle d'isolation mémoire dynamique à base de MMU

Narjes Jomaa & David Nowak & Gilles Grimaud & Julien Iguchi-Cartigny

Laboratoire d'Informatique Fondamentale de Lille
CNRS et Université Lille 1

La protection des données utilisées par un logiciel vis-à-vis d'un accès malveillant ou simplement malvenu est un enjeu qui fut identifié dès les premiers développements de l'informatique. Différentes approches ont été explorées depuis lors. Ce que fait un logiciel est défini par son code, exprimé dans un langage de programmation. Dès les années 1970, la première approche a donc consisté à étendre les langages de programmation avec des éléments permettant de contrôler l'accès et l'utilisation des informations, au niveau du code source (et typiquement lors de la compilation des programmes). La littérature regroupe ces techniques d'isolation en termes de confidentialité et d'intégrité, sous la mention générique d'isolation statique par le langage. Cependant en 2004, Steve Zdancewic [7] constatait que ces langages n'avaient toujours pas trouvé leur public et que le plus grand défi des outils existants était, aujourd'hui encore, d'en démontrer la pertinence industrielle. Mais durant les années 70, une seconde approche [3], fondée sur le contrôle d'accès à la mémoire matérielle pendant l'exécution, à l'aide de matériels dédiés, a connu un essor bien plus important. Elle assure, depuis lors, l'isolation des données sur nos stations de travail et nos serveurs. La littérature parle d'isolation dynamique par adressage. Cependant cette solution n'a pas connu les mêmes efforts de formalisation que la première. L'isolation mémoire du micronoyau seL4 [5] se fonde sur cette approche, mais pas celle des programmes qui y sont ensuite déployés. Nous nous proposons ici d'étudier les éléments de formalisation nécessaires à la preuve d'isolation de la mémoire entre processus concurrents exécutés par un système d'exploitation sur un matériel conventionnel. L'isolation est en règle générale assurée par le logiciel et le matériel à l'aide d'un mécanisme de gestionnaire de la mémoire (MMU ou *Memory Management Unit*). Ce matériel est en charge d'une part de réserver des pages physiques et d'autre part de la gestion des exceptions liées à la mémoire. Au niveau logiciel, un service commande le MMU afin d'assurer l'isolation des processus dans leurs pages mémoires physiques dédiées. À l'aide de l'assistant de preuve Coq nous avons formalisé un système d'exploitation minimaliste tournant sur un matériel conventionnel et avons en partie prouvé que le système est capable d'interdire les accès illégaux vers la mémoire.

Travaux reliés Il existe plusieurs projets qui s'intéressent à l'isolation de la mémoire s'appuyant sur un composant matériel de la gestion mémoire (MMU). seL4 [5] est un micronoyau L4 construit à partir d'un modèle en Isabelle possédant des preuves de fonctionnement cohérent et la démonstration que le binaire produit par un compilateur C répond au modèle de la spécification du micronoyau. Néanmoins l'ensemble des preuves est construit sur l'hypothèse de la confiance sur le matériel (MMU, périphériques, etc.). De plus, les développeurs ont choisi d'implémenter le gestionnaire mémoire des processus à l'extérieur du micronoyau. Par conséquent, seL4 ne garantit que l'isolation de la mémoire du noyau par rapport aux autres processus et ne gère pas l'isolation entre processus. Enfin, le modèle du gestionnaire des interruptions n'est pas réaliste, dû au fait qu'en mode noyau les interruptions sont généralement désactivées.

Verisoft est un autre projet de recherche élaboré par le centre aérospatial d'Allemagne qui vise la vérification formelle de plusieurs systèmes informatiques dans les domaines de l'automobile, de la sécurité informatique, et des instruments médicaux. Parmi leurs travaux PikeOS [2] est un micronoyau dérivé de L4. Sans compter la vérification d'un simple appel système qui permet de changer la priorité des threads aucune autre propriété n'a été prouvée. L'outil de preuve utilisé est VCC développé par Microsoft.

BabyVMM [6] est un gestionnaire de mémoire fourni par le micronoyau CertiKOS [4]. Il s'agit d'une implémentation simplifiée des services d'allocation de mémoire et de traduction d'adresse. Parmi les couches de BabyVMM se trouve l'allocateur de la mémoire et le gestionnaire des tables de pages. Chaque couche est conçue d'une façon modulaire qui facilite sa réutilisation et son extension. Le gestionnaire des tables de pages est basé sur le mécanisme de traduction d'adresse. Vu sa complexité, les auteurs ont implémenté un module abstrait, plus simple, appelé AS (*Address Space*) qui considère la mémoire physique comme un espace d'adressage virtuel plus grand que la mémoire physique et qui est vu sous la forme des pages allouées ou non. Ainsi, la certification de la BabyVMM est un résultat immédiat des preuves de raffinement sans avoir besoin de preuves supplémentaires. Mais le système d'allocation de la mémoire est très faible côté performances car il effectue une recherche exhaustive des pages libres.

Barthe et al. [1] ont également formalisé un hyperviseur en Coq appelé VirtualCert. Il s'agit d'une spécification exécutable exprimée avec une sémantique axiomatique s'appuyant sur des pré-conditions et post-conditions. Dans ce modèle, ils ont mis en place un mécanisme de pagination et ils ont prouvé certaines propriétés d'isolation entre les machines virtuelles qui s'exécutent au-dessus de l'hyperviseur formalisé.

Contributions Notre objectif est de traiter les problématiques qui ne sont pas prises en compte dans la formalisation de seL4 bien qu'elles soient fondamentales pour la sécurité d'un système d'exploitation. Nous avons donc formalisé en Coq un système d'exploitation minimaliste qui fournit un gestionnaire de mémoire plus performant que celui de CertiKOS, du point de vue mécanisme d'allocation de pages mémoire, et qui permet une allocation dynamique des pages mémoire en se basant sur le mécanisme de la virtualisation fourni par le MMU. L'algorithme d'ordonnancement des processus que nous avons formalisé est préemptif et s'appuie sur les interruptions matérielles contrairement à seL4 qui désactive généralement les interruptions en mode noyau et dont l'ordonnanceur est coopératif. Plus précisément, nous avons formalisé en Coq une logique de Hoare au-dessus d'une monade d'état nous permettant de spécifier les propriétés souhaitées, et prouvé des lemmes fondamentaux qui sont nécessaires pour faire des preuves avec cette logique. Puis nous avons formalisé l'architecture matérielle et l'intégralité des services fournis par notre système d'exploitation minimaliste et avons en partie prouvé la propriété d'isolation mémoire dynamique.

Modélisation de MMU Nous avons modélisé, en Coq, le comportement de MMU sur lequel repose notre mécanisme d'isolation mémoire. En effet, l'adresse virtuelle est décomposée en deux parties : `index` et `offset`. `index` correspond à l'index à considérer dans la table de pages pour faire la traduction et `offset` correspond à la position au sein de la page. Les entrées dans cette page s'appellent des PTE (*Page Table Entry*). Chaque PTE, référençant une page, contient la base de l'adresse physique associée et les bits de contrôle d'accès. Le nombre de bits utilisé pour représenter le numéro de la page physique est défini par un entier. Les bits de contrôle sont composés de deux valeurs. La première défini si la page mémoire virtuelle existe ou pas. La seconde défini si le processus y a légitimement accès ou pas. L'opération `translate` réalise le calcul de traduction d'adresse. Cette opération utilise la table de pages du processus courant pour trouver une correspondance entre l'adresse passée en paramètre et une adresse physique. Si l'entrée dans la table de pages, c.-à-d. le PTE, n'est pas valide alors elle retourne une exception `faultpage`. Si le PTE est valide, alors elle contrôle les droits d'accès. Si, dans le PTE, les droits d'accès sont insuffisants, alors elle retourne une exception `noaccess` sinon elle retourne

l'adresse physique associée à l'adresse virtuelle. Cette opération `translate` de notre modèle est dans la pratique implémentée physiquement dans l'architecture.

Formalisation du système d'exploitation Pour formaliser le système d'exploitation, nous spécifions dans un premier temps son état interne par le biais d'un ensemble de structures exprimées en Coq. Plus précisément, la structure d'un processus et la structure qui représente une vue globale sur les différents éléments qui constituent l'état de notre système et son architecture. Cette dernière regroupe principalement la liste des processus inactifs, le processus en cours d'exécution et son état, la liste des instructions du système d'exploitation et de tous les processus lancés, la liste des interruptions, la mémoire physique du système, la première page libre, etc. Ces structures sont utilisées par l'opération `step` qui modélise l'exécution d'un cycle (autrement dit, un pas) de l'architecture et le changement de son état au cours de ce cycle. À chaque pas, cette opération vérifie s'il y a une interruption ou non. Si il y a une interruption elle déclenche son traitement, sinon elle récupère la prochaine instruction à exécuter. Au moment de l'initialisation du système, la mémoire est initialisée afin de gérer facilement les pages libres. En effet, les premiers octets de chaque page sont initialisés par la position de la première page libre suivante et les premiers octets de la dernière page libre sont initialisés par la valeur du nombre de pages. Cette façon de gérer les pages libres est simple tout en étant raisonnablement efficace. Elle nous permet en plus de gagner de l'espace mémoire vu que nous utilisons l'espace libre lui-même pour gérer les pages libres. Le mécanisme d'allocation de pages se fonde sur une opération qui modifie la première page libre de l'état globale du système par la page libre suivante. Cette opération est utilisée par la fonction de création d'un processus en lui allouant deux pages, une pour sa table de pages et une autre pour son espace de travail. Cette opération modifie la mémoire en ajoutant une entrée dans la table de pages relative à la deuxième page allouée et modifie l'état globale du système en ajoutant le nouveau processus à la liste des processus lancés par le système.

Ordonnancement Le processus d'ordonnancement des tâches s'appuie sur les interruptions. En effet, à chaque pas d'exécution, nous vérifions dans la liste des interruptions s'il existe une interruption à exécuter, sinon nous passons à l'instruction suivante. Si l'interruption référence l'opération `Switch_proces` alors nous sauvegardons l'état du processus courant puis nous choisissons le premier processus dans la liste des processus, de l'état courant, comme nouveau processus courant. À la fin de ces opérations le MMU est reconfiguré de telle sorte qu'il référence la table de pages du nouveau processus élu. Notez que l'instruction `Switch_process` est en fait une sorte de méta-instruction que nous avons introduite dans notre modèle d'architecture pour le simplifier dans un premier temps. Cette méta-instruction se décompose en réalité en plusieurs instructions plus primitives et s'exécute par le noyau pour assurer la préemption de l'ordonnancement.

Preuve formelle d'isolation en Coq Après la définition en Coq de tous les services fournis par le système, nous avons défini pour chaque opération, utilisée par la preuve ultérieurement, le triplet qui lui correspond et nous avons prouvé qu'il est correct. L'étape suivante a consisté à formaliser et prouver l'isolation dynamique de la mémoire en Coq. En effet, nous avons commencé par définir la propriété d'un état cohérent d'un système préservant l'isolation de la mémoire. Pour définir un tel état, nous nous sommes appuyés sur les propriétés fondamentales de notre modèle de MMU sur lequel repose notre mécanisme d'isolation mémoire. Nous avons appelé cette définition `Isolated` et nous l'avons résumée de la manière suivante : « si nous prenons deux processus différents `p1` et `p2` ayant chacun une référence vers une table de pages différente alors toute adresse référencée par la table de pages du processus `p1` doit être différente de toute adresse référencée par la table de pages du processus `p2` ». Notre définition dans son état actuel est plus faible que ce qu'on pourrait souhaiter car elle ne prend en compte que le cas où tous les processus ont des tables de pages différentes. Elle est cependant, dans ce cas, suffisante pour garantir l'isolation de la mémoire entre les processus. En

effet, tout accès à la mémoire doit passer par la table de pages du processus concerné et donc une construction valide de la table de pages garantit un état cohérent du système.

Nous devons prouver par la suite que la construction des tables de pages des processus préserve l'isolation mémoire. D'après la définition d'isolation nous avons conclu que afin de garantir l'isolation de la mémoire, il faut prouver que l'allocateur de pages n'alloue que des pages libres (c'est-à-dire des pages non référencées par un autre processus). Pour cela, nous devons prouver que l'opération de création des processus préserve l'isolation parce que cette fonction crée la table de pages du processus et lui alloue quelques pages de travail. Donc, l'étape suivante est de prouver que la propriété d'isolation reste valide après la création d'un processus. Cette opération est appelée `create_process`. Son triplet de Hoare dépend de l'état courant `s` du système et d'un prédicat `P` quelconque qui représente la post-condition de ce triplet. La création d'un processus devant garantir l'isolation de la mémoire, nous appliquons ce triplet en instanciant sa post-condition `P` avec notre prédicat d'isolation `Isolated`. Nous avons défini une fonction `get_allocated_pages` qui retourne toutes les pages déjà allouées dans un état donné `s`. Cette fonction n'est utilisée que pour la production de la preuve. Comme résultat nous avons partiellement prouvé la propriété d'isolation suivante :

```
Lemma prop_isolation (addr :nat) : ∀ s,
 Isolated s /\ snd_page_isValid s /\ first_page_isValid s /\
 ~ In (first_allocated_page s) (get_allocated_page s.(process_list) nil s (data)) /\
 ~ In (snd_allocated_page s) (get_allocated_page s.(process_list) nil s.(data)) /\
 first_allocated_page s <> snd_allocated_page s →
 verified_create_process_wp(λ  addr s ⇒ Isolated s) addr s.
```

Ce lemme consiste à prouver qu'à partir d'un état `s` qui préserve l'isolation et tel qu'il y a encore de l'espace mémoire libre pour les deux pages à allouer pour le processus, et si ces deux pages sont différentes et n'appartiennent pas à la liste des pages retournées par la fonction `get_allocated_pages` alors l'opération `create_process` préserve l'isolation. Autrement dit, la construction de la table de pages est valide et garantit un état cohérent du système préservant l'isolation de la mémoire.

Références

[1] G. Barthe, G. Betarte, J. D. Campo, J. M. Chimento, and C. Luna. Formally verified implementation of an idealized model of virtualization. *19th International Conference on Types for Proofs and Programs (TYPES 2013)*, 26 :45–63, 2014.

[2] C. Baumann, B. Beckert, H. Blasum, and T. Bormer. Ingredients of operating system correctness, lessons learned in the formal verification of PikeOS. *Emb. World Conf., Nuremberg, Germany*, 2010.

[3] D. E. Bell and L. J. La Padula. Secure computer system : Unified exposition and Multics interpretation. Technical report, DTIC Document, 1976.

[4] L. Gu, A. Vaynberg, B. Ford, Z. Shao, and D. Costanzo. CertiKOS : A certified kernel for secure cloud computing. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, APSys '11, pages 3 :1–3 :5, New York, NY, USA, 2011. ACM.

[5] G. Klein, J. Andronick, K. Elphinstone, T. Murray, T. Sewell, R. Kolanski, and G. Heiser. Comprehensive formal verification of an OS microkernel. *ACM Transactions on Computer Systems*, 32(1) :2 :1–2 :70, feb 2014.

[6] A. Vaynberg and Z. Shao. Compositional verification of a baby virtual memory manager. In C. Hawblitzel and D. Miller, editors, *Certified Programs and Proofs*, volume 7679 of *Lecture Notes in Computer Science*, pages 143–159. Springer Berlin Heidelberg, 2012.

[7] S. Zdancewic. Challenges for information-flow security. *Proceedings of the 1st International Workshop on the Programming Language Interference and Dependence (PLID'04)*, 2004.