La conception d'un noyau orientée par sa preuve d'isolation mémoire

Narjes Jomaa, Samuel Hym, David Nowak

▶ **To cite this version:**

Narjes Jomaa, Samuel Hym, David Nowak. La conception d'un noyau orientée par sa preuve d'isolation mémoire. Compas 2018, Jul 2018, Toulouse, France. hal-01819955

HAL Id: hal-01819955
https://hal.science/hal-01819955v1

Submitted on 7 Sep 2018

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

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

Compas'2018 : Parallélisme/ Architecture / Système
Toulouse, France, du 03 au 06 juillet 2018

La conception d'un noyau orientée par sa preuve d'isolation mémoire *

Narjes Jomaa, Samuel Hym, David Nowak

CRIStAL, CNRS & Université de Lille

Résumé
La vérification des logiciels critiques tels que les noyaux des systèmes d'exploitation est une activité longue et complexe. Celle-ci nécessite une attention particulière dès les premières phases de conception. Le choix des fonctionnalités, la stratégie d'implémentation, le modèle du matériel à gérer, les propriétés à vérifier et l'approche de vérification sont tous à prendre en compte pendant le développement du noyau. Ce papier discute de la stratégie de *co-design* d'un noyau de système d'exploitation et de sa preuve. L'objectif visé est d'adapter le design du noyau pour réduire la complexité de la production de sa preuve tout en préservant son utilisabilité. La propriété étudiée dans ce papier est une propriété de sécurité, exprimée en terme d'isolation mémoire. L'étude présentée met notamment en évidence l'importance de l'interface d'accès au matériel. *In fine* c'est cette interface qui est axiomatisée pour construire la preuve du noyau.

Mots-clés : preuve formelle, co-design, isolation mémoire, Pip

1. Introduction

La sécurité des logiciels critiques est aujourd'hui un enjeu majeur dans le monde informatique. De nouvelles attaques confirment chaque jour la nécessité d'investir plus dans les outils et les techniques qui permettent de réduire la vulnérabilité de nos systèmes. Le plus souvent il suffit d'exploiter des failles dans le système d'exploitation pour compromettre la sécurité globale d'un système. Il existe plusieurs travaux autour de la vérification formelle des systèmes d'exploitation. Cette technique est fondée sur des notations mathématiques permettant de raisonner rigoureusement sur les programmes. Une grande majorité de ces travaux consiste à prouver des propriétés sur des systèmes de taille importante ce qui rend l'étape de la maîtrise de tous les détails de ce logiciel critique assez longue et donc la vérification devient plus compliquée à établir. Prenons l'exemple du noyau seL4 [9] qui est aujourd'hui considéré comme étant le micro-noyau qui a les résultats les plus satisfaisants au niveau des preuves. L'objectif principal est de vérifier le bon fonctionnement du système ainsi que quelques propriétés de sécurité. seL4 contient environ 8700 lignes de C et 600 lignes d'assembleur impliquant environ 480 000 lignes de preuve en Isabelle/HOL [16]. L'hyperviseur HyperV [12] contient environ 100 Kloc de C et 5 Kloc d'assembleur dont seulement 20% a été vérifié [15]. Le noyau µC/OS-II [29], dont le bon fonctionnement de certains modules a été formellement vérifié, contient environ 6400 de lignes de C et d'assembleur. CertiKOS [5] est un autre noyau tel que le mécanisme de concurrence a été vérifié. Il contient 6500 lignes de C et d'assembleur ainsi que 450 lignes pour spécifier le modèle abstrait du noyau. Dans ce papier nous discutons nos choix de conception et de vérification

*. Ce travail a été financé par le projet européen Celtic-Plus ODSI C2014/2-12

du proto-noyau Pip [8, 27] qui ont favorisé la réduction de la taille du code du noyau ainsi que le coût de la production de sa preuve. En effet, la sécurité assurée par ce noyau est fondée sur le principe de partitionnement de la mémoire et les services fournis sont écrits en Gallina, le langage de spécification de Coq [26], et traduit automatiquement vers C en utilisant Digger [22].

2. Que cherche-t-on à garantir ?

Le choix des propriétés à vérifier dépend fortement du contexte dans lequel le système sera utilisé. Il est possible par exemple de prouver que les programmes se comportent correctement. Autrement dit établir une preuve mathématique sur le comportement fonctionnel du système. En revanche, même un système qui fonctionne correctement est susceptible de contenir des bugs de sécurité ce qui rend les programmes facilement attaquables. Dans le contexte des logiciels critiques tels que les noyaux des systèmes d'exploitation, il est prioritaire de prouver des propriétés de sécurité plutôt que de se concentrer sur le bon fonctionnement. Le choix des propriétés fonctionnelles dépend fortement des détails d'implémentation du système. Donc si le système est très complexe il est nécessaire de définir et vérifier de nombreuses propriétés pour prendre en compte tous les comportements possibles prévus par ce système, ce qui risque de faire apparaître un nombre important des propriétés. En effet, la sécurité n'est pas une propriété fonctionnelle et donc elle n'est pas liée à une fonctionnalité en particulier. Ainsi, sa preuve consiste à définir formellement ce qu'est cette propriété de sécurité puis prouver qu'elle est préservée par chaque étape de l'exécution du système.
La sécurité, également, est un terme assez général. Dans le contexte d'un système d'exploitation la sécurité, telle que identifiée par Rushby [20, 19], consiste principalement à assurer la séparation entre les entités ainsi que contrôler les communications entre elles. Une première étape dans ce processus de vérification nécessite donc de définir clairement les propriétés de sécurité visées par la preuve et d'identifier les différents composants matériels ou logiciels sur lesquels reposent ces propriétés. Dans ce contexte nous nous intéressons à la propriété d'isolation mémoire. Cette propriété permet d'assurer qu'un programme ne peut pas accéder à la mémoire d'un autre programme. Nous considérons que c'est la propriété de sécurité la plus fondamentale car elle permet de prouver d'autres propriétés telles que celles qui consistent à assurer la sécurité de la communication entre les processus.
Il est important de noter que d'après notre expérience, la vérification des propriétés de sécurité nous amène également à prouver certaines propriétés sur le bon fonctionnement. Elles sont formalisées soit sous la forme de propriétés de cohérence [7] ou encore définies comme propriétés sur l'évolution de l'état au sein des services noyau. En revanche nous nous sommes limités à celles qui sont nécessaires pour prouver les propriétés de sécurité. Cela permet d'identifier les propriétés fonctionnelles les plus pertinentes tout en assurant la vérification des propriétés fondamentales de sécurité.

3. Méthodologie de développement et choix de conception de Pip

Une méthodologie traditionnelle de développement d'un noyau et sa vérification consiste à suivre d'une manière séquentielle les étapes suivantes : Conception, implémentation et preuve. En revanche, le *co-design* de noyau avec sa preuve nous amène à adapter cette approche en autorisant des *feedbacks* entre ces étapes. Autrement dit, influencer le développement d'une étape par une autre pour réduire le coût de la preuve et assurer l'utilisabilité du système. En effet, pour faciliter à la fois la preuve et la traduction du code, l'implémentation de Pip se décompose principalement en deux couches.

API de Pip La première couche est la partie du code convertible de Gallina vers C. Elle comprend l'implémentation des différents services fournis par Pip. Ces services gèrent principalement la mémoire virtuelle des partitions et les interruptions, et assurent l'isolation mémoire. Il s'agit de dix appels systèmes [8] choisis minutieusement pendant la phase de conception afin d'assurer à la fois la faisabilité de la preuve et l'utilisabilité du système. Ce choix de minimisation du TCB (Trusted Computing Base) est principalement motivé par la réduction du coût de la preuve. Cependant, la réduction de la taille du TCB n'est pas un concept inconnu. Il a été adopté par la famille des micro-noyaux [13]. Il a été montré qu'il est possible de réduire le code exécuté en mode noyau à la gestion de la mémoire virtuelle, l'ordonnancement, la communication entre les processus, le multiplexage et le changement de contexte. Le reste des modules tels que la gestions des fichiers et les périphériques peuvent s'exécuter au même niveau que les applications utilisateur sans avoir d'impact sur la sécurité. Les exonoyaux [4] sont conçus en suivant le même principe. En effet, ces travaux ont montré qu'il est possible de réduire encore plus le nombre de fonctionnalités exécutées en mode noyau. Leur modèle de sécurité exige d'implémenter la gestion de la mémoire virtuelle, le changement de contexte et le multiplexage en mode privilégié. De façon similaire, le *co-design* du proto-noyau Pip avec sa preuve (i.e. influencer la conception pour faciliter la preuve) nous a menés à réduire le TCB au strict minimum en exportant tous les modules qui ne sont pas nécessaires pour la sécurité en mode utilisateur, y compris le multiplexage.

La gestion de la mémoire assurée par Pip est fondée sur le modèle de partitionnement hiérarchique [21]. Au démarrage du système, toute la mémoire physique disponible est mappée dans la partition racine. Cette dernière, comme n'importe quelle nouvelle partition, est autorisée à créer des partitions enfants en utilisant uniquement sa propre mémoire pour construire un modèle arborescent. Chaque nouvelle partition est configurée de sorte que toutes les pages associées à cette partition sont mappées dans la partition parent (ce qu'on appelle le *partage vertical*). Cette structure arborescente nécessite des structures supplémentaires en plus des pages du MMU dans chaque partition afin de garder des informations sur chaque page associée à une partition. Le choix de ces structures est également motivé par la réduction du coût de la preuve et l'utilisabilité du système. En effet, une première structure que nous avons appelée *shadow1* reprend la structure arborescente des pages de configuration du MMU pour y stocker d'autres informations. Cette structure est nécessaire pour assurer l'isolation mémoire. Les données stockées dans cette structure permettent au noyau de savoir si une page mappée dans une partition parent a été partagée avec une partition fille ou non. Cela évite de mapper la même page dans plusieurs enfants différents car dans ce cas l'isolation mémoire ne serait plus garantie. Deux autres structures *shadow2* (qui a également la même structure arborescente du MMU) et *linkedList* (une liste de paires d'adresses virtuelles et physiques) ont été mises en place pour faciliter la remise des pages à la partition parent après leur récupération dans le fils. Contrairement au *shadow1* ces deux dernières structures ne sont pas nécessaires pour la sécurité mais elles sont utiles pour améliorer la performance de l'exécution des appels système fournis par le noyau.

À cet égard Pip gère uniquement la mémoire des partitions (y compris la configuration du MMU et les structures internes des partitions) et le changement de contexte (qui se résume simplement à un basculement du flot d'exécution soit vers la partition racine, soit vers le père de la partition qui est en train de s'exécuter ou bien vers l'un de ses fils).

Tous les services du noyau sont écrits dans un style impératif grâce à une monade d'état [28] qui permet d'introduire et gérer facilement toute sorte d'effet de bord tels que la modification de l'état du système. L'une des particularités de notre stratégie d'implémentation est qu'aucun objet de la librairie standard de Coq tels que les listes ou les arbres n'ont été utilisés dans l'implémentation des services. En effet, de point de vue preuve il est plus simple de raisonner

sur ce type de structures abstraites, en outre, cela peut complexifier la preuve du traducteur de Gallina vers C. Ainsi, comme réponse à ce problème, toutes les structures maintenues par le noyau (listes et arbres) sont codées explicitement dans la mémoire physique. Cela exige uniquement des primitives assurant la lecture et l'écriture dans la mémoire à des adresses physiques bien définies. Cet ensemble constitue le HAL.

HAL La deuxième couche contient quelques dizaines de primitives assez simples et auxquelles nous faisons confiance. Elles sont écrites directement en C et en assembleur et uniquement utilisées par le noyau pour accéder à l'état du matériel. Pour chacune de ces primitives nous avons défini une fonction en Gallina modélisant le comportement de celle-ci. Par exemple, une des primitives en HAL est simplement un accès en lecture ou en écriture à la mémoire physique. Autrement dit, c'est la partie du code qui ne peut pas être exprimée en Gallina. Concrètement, la mémoire physique est modélisée par une liste d'associations en Gallina, telle que la clé est une adresse physique et la valeur est la donnée sauvegardée à cette adresse. L'accès matériel permettant la modification de la mémoire physique en C est modélisé par la modification de la liste d'associations.

L'exemple suivant correspond à une fonction interne de Pip (`getFstShadow`), écrite en Gallina et sa traduction *mot à mot* en C. Cette fonction permet de récupérer une référence vers une structure de configuration d'une partition. Elle consiste en une séquence de trois primitives HAL : `getSh1idx`, `succ` et `readPhysical` et retourne la référence en question.

Le code Gallina (à gauche) sera ainsi automatiquement traduit dans le code C (à droite) :

```
Definition getFstShadow part :=
  perform idx := getSh1idx in
  perform idxSucc := Index.succ idx in
  readPhysical part idxSucc.
```

```
uintptr_t getFstShadow(uintptr_t part) {
  const uint32_t idx = getSh1idx();
  const uint32_t idxSucc = succ(idx);
  return readPhysical(part, idxSucc); }
```

Il est important de noter qu'une primitive HAL ne correspond pas à une instruction du CPU. Elle consiste simplement en une opération élémentaire, du point de vue du modèle, permettant d'accéder à la mémoire, effectuer un calcul logique ou alors récupérer la valeur d'un paramètre lié à l'architecture.

4. Confiance en HAL

Concrètement, le comportement du matériel est plus compliqué que notre modèle abstrait de HAL en Gallina. En plus de la mémoire physique, il peut être nécessaire de gérer d'autres composants internes tels que les caches ou le TLB. Nous considérons que le HAL représente une abstraction du comportement du matériel géré par le noyau. C'est une manière de formaliser l'idée que le développeur se fait du comportement du matériel tout en gardant une formalisation indépendante d'une architecture en particulier.

Nous partons du principe que la vérification des propriétés de sécurité nécessite dans un premier temps d'identifier clairement le rôle du noyau et celui du matériel dans le contexte de la sécurité. Cela nous permet de découpler l'implémentation du noyau des hypothèses sur lesquelles reposent les propriétés de sécurité. Le but de nos travaux est de vérifier les propriétés du noyau, donc la manière dont le matériel effectue sa tâche devient un problème orthogonal et nécessite une preuve complémentaire [2, 25, 11, 14]. Par exemple, Pip contrôle l'accès des processus à la mémoire physique à travers le MMU. Donc nous prouvons que la propriété de sécurité est préservée à travers une configuration correcte des tables de MMU. Cette configuration est effectuée par le noyau en s'appuyant sur l'hypothèse que le MMU traduit correctement les adresses virtuelles en adresses physiques. De même, la configuration des structures internes des

partitions nécessite des opérations d'écriture dans la mémoire physique. Le rôle du matériel ici est d'effectuer la modification, en revanche c'est le noyau qui doit fournir les valeurs. Par conséquence, le but principal de la vérification sera plutôt sur les valeurs fournies par le noyau en fonction desquelles le matériel effectue l'opération. L'exemple suivant permet de comparer l'implémentation en C d'une primitive HAL avec son modèle en Gallina. Cette primitive, `readPhysical`, retourne l'adresse physique sauvegardée à l'adresse passée en paramètre. Dans le contexte de `getFstShadow`, la valeur retournée par cette primitive correspond à une valeur dans une page de configuration d'une partition. En C, il s'agit simplement de récupérer la valeur sauvegardée dans `table` à la position `idx`. Par contre, son modèle en Coq, qui est une fonction monadique, met en place un test supplémentaire permettant de vérifier que le noyau avait déjà sauvegardé une valeur de type *page* (i.e. PP) à cette adresse physique sinon c'est considéré comme un comportement indéfini. Ainsi, pour prouver que cette primitive retourne bien un numéro d'une page physique, il suffit de prouver que les arguments fournis par le noyau ont les propriétés nécessaires pour satisfaire le test défini dans le modèle en Coq. Donc concrètement, nous supposons que l'implémentation de `readPhysical` en C est correcte mais nous prouvons que les valeurs `table` et `idx` fournies par le noyau ne produisent pas un comportement indéfini.
La primitive `readPhysical` en C :

```
uint32_t readPhysical(uint32_t table, uint32_t idx) {                    1
    disable_paging();  /* En mode noyau : nous pouvons désactiver le MMU */  2
    uint32_t dest = table | (idx * sizeof(uint32_t));                    3
    uint32_t val = *(uint32_t *) dest;              /* Récupère l'entrée */  4
    enable_paging();                                /* Réactive le MMU */    5
    return val & 0xFFFFF000;                        /* Retourne la valeur */ 6
}                                                                        7
```

Son modèle en Gallina :

```
Definition readPhysical (paddr : page) (idx : index) : LLI page :=          1
  perform s := get in                                          (* Récupère l'état *)     2
  let e := lookup paddr idx s.(memory) beqPage beqIndex in (* Récupère l'entrée *)   3
  match e with                                                               4
   | Some (PP a) ⇒ ret a          (* Retourne la valeur si elle est de type Page *)  5
   | _           ⇒ undefined 5    (* Sinon signale un comportement indéfini *)       6
  end.                                                                       7
```

5. Quelle approche de vérification ?

Pour prouver des propriétés sur leurs systèmes, la majorité des travaux [10, 5, 18, 3, 23] utilise une approche par raffinement [1]. Cette approche consiste à développer et prouver des systèmes de manière incrémentale. Il s'agit de définir un modèle abstrait du système désiré, puis à partir de ce dernier définir graduellement son implémentation en introduisant à chaque niveau plus de détails sur les objets manipulés par ce système tels que les structures de données et les variables. La preuve de la propriété de sécurité se fait uniquement au niveau du modèle abstrait. Chaque niveau inférieur est considéré comme un raffinement des niveaux précédents et une preuve est nécessaire pour prouver la validité de chaque niveau de raffinement.
Cette approche est avérée la moins coûteuse en supposant que sa preuve est plus rapide à établir en la comparant avec l'établissement de la preuve des propriétés directement sur le code. Mais cela a aussi un coût. En effet, la définition d'un modèle abstrait de ce qu'on vise à implémenter repose généralement sur une intuition de ce que fait le système [17]. Donc le risque de définir un comportement abstrait qui n'existe pas au niveau de l'implémentation concrète est tout à fait

possible. Plus précisément, nous pouvons définir des comportements qui n'existeront pas dans le vrai système. Puis pendant l'étape de vérification des raffinements il est possible de détecter cette incohérence dans un niveau plus loin voire seulement au niveau de l'implémentation finale. Dans ce cas il faut modifier toutes les spécifications intermédiaires ainsi que la preuve de la validité du raffinement.

Le *co-design* d'un noyau et de sa preuve nécessite des échanges réguliers entre les deux équipes de développement et de vérification dans le but d'obtenir un compromis entre l'utilisabilité du logiciel conçu et la faisabilité de sa preuve. Mais, le processus de raffinement, en pratique, est généralement dirigé par l'équipe de vérification et rarement maîtrisé par les développeurs, ce qui limite la collaboration entre les deux équipes [24, p.xxiii]. Il est donc possible de faire des choix de design qui ne seront pas validés pendant la dernière phase de conception (c'est-à-dire l'implémentation finale). Le noyau Pip que nous avons développé est plus petit (à dessein) que les noyaux identifiés au début de ce papier, il contient environ 1500 lignes de code et comprend les services nécessaires pour assurer la sécurité et son utilisabilité. La réduction de la taille du code à prouver nous a menés à adapter la méthodologie de vérification. En effet, nous avons appliqué une approche différente qui est la preuve directement sur le code en s'appuyant sur les invariants [7]. L'exemple suivant consiste à prouver le triplet de Hoare [6] définissant l'invariant d'un appel système *sys_call* : {isolation & cohérence} *sys_call* {isolation & cohérence}.

sys_call pourrait être l'un des services fournis par le noyau Pip. Notre processus de vérification consiste à prouver que les propriétés d'isolation et de cohérence sont préservées par tous les services du noyau (la formalisation des propriétés et leurs vérification ne font pas l'objet de ce papier : voir [8] pour plus de détails) et que la traduction de Gallina vers C est correcte.

Il est important de noter que prouver des propriétés directement sur le code donne au *prouveur* la maîtrise de tous les détails *d'implémentation*. Suite à notre expérience, cela s'est avéré important pour faciliter le raisonnement sur certaines séquences d'instructions. Bien que, du point de vue concepteur c'est le résultat final de l'exécution d'un service noyau qui compte, l'ordre dans lequel les instructions sont exécutées peut compliquer la preuve. Dans ce contexte nous avons identifié deux règles : rapprocher le calcul d'une valeur de son utilisation et éviter les incohérences temporaires dans les structures de données d'une partition pendant l'exécution d'un service. Par exemple, considérons une fonction qui met à jour une liste chaînée, quand il s'agit d'ajouter d'un élément dans cette liste il est recommandé d'initialiser la nouvelle cellule avec la bonne valeur avant de l'associer à la liste. Cela garantit la cohérence durant l'exécution de la fonction. Ainsi, à plusieurs reprises nous avons décidé de changer l'ordre d'exécution de certaines instructions pour faciliter l'établissement de notre preuve.

Invariants	**lignes de preuve**	**temps**
createPartition ($\approx$ 300loc)	$\approx$ 60000	$\approx$ 10 mois
createPartition + addVaddr ($\approx$ 110loc)	$\approx$ 78000	$\approx$ 2 mois
createPartition + addVaddr + mappedInChild ($\approx$ 40loc)	$\approx$ 78300	$\approx$ 4 heures

TABLE 1 – L'organisation de la preuve

Actuellement une partie importante de l'implémentation du noyau a été vérifiée. Elle concerne les services illustrés par le tableau 1. La vérification de l'appel système `createPartition` correspond à la preuve à laquelle nous avons consacré le plus d'effort. Cela est justifié d'une part par sa complexité (il consiste en environ 300 lignes de code), et d'autre part parce que c'était le premier service abordé par la vérification. En effet, durant le processus de sa preuve, nous avons défini la majorité des propriétés de cohérence et donc une partie importante des théorèmes sur les structures manipulées par le noyau ont été définis et prouvés.

6. Conclusion

Nous avons montré que le *co-design* de noyau avec sa vérification est important pour faciliter sa preuve. Des choix de conception nécessitent d'être étudiés afin de trouver un compromis entre la faisabilité de la preuve et l'utilisabilité du système. Ces choix concernent principalement le fait de placer des fonctionnalités noyau en mode utilisateur tout en garantissant la propriété d'isolation mémoire. Dans la suite de nos travaux, nous souhaitons étendre le modèle de HAL pour formaliser et vérifier les mécanismes matériels tels que la gestion des caches et de TLB.

Bibliographie

1. Abadi (M.) et Lamport (L.). – The existence of refinement mappings. *Theoretical Computer Science*, 1991, pp. 253–284.
2. Barthe (G.), Betarte (G.), Campo (J. D.) et Luna (C.). – Cache-leakage resilient OS isolation in an idealized model of virtualization. – In *Computer Security Foundations Symposium (CSF), 2012 IEEE 25th*, pp. 186–197. IEEE, 2012.
3. Dam (M.), Guanciale (R.), Khakpour (N.), Nemati (H.) et Schwarz (O.). – Formal verification of information flow security for a simple ARM-based separation kernel. – In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pp. 223–234, 2013.
4. Engler (D. R.), Kaashoek (M. F.) et O'Toole, Jr. (J.). – Exokernel : An operating system architecture for application-level resource management. – In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, *SOSP*, SOSP, 1995.
5. Gu (R.), Shao (Z.), Chen (H.), Wu (X. N.), Kim (J.), Sjöberg (V.) et Costanzo (D.). – CertiKOS : An extensible architecture for building certified concurrent OS kernels. – In *OSDI*, pp. 653–669, 2016.
6. Hoare (C. A. R.). – An axiomatic basis for computer programming. *Commun. ACM*, vol. 12, n 10, 1969, pp. 576–580.
7. Jomaa (N.), Nowak (D.), Grimaud (G.) et Hym (S.). – Formal proof of dynamic memory isolation based on MMU. *Science of Computer Programming*, 2017.
8. Jomaa (N.), Torrini (P.), Nowak (D.) et Grimaud (G.). – Proof-oriented design of a separation kernel with minimal trusted computing base. – In *18th International Workshop on Automated Verification of Critical Systems*, 2018.
9. Klein (G.), Andronick (J.), Elphinstone (K.), Murray (T.), Sewell (T.), Kolanski (R.) et Heiser (G.). – Comprehensive formal verification of an OS microkernel. *ACM Trans. Comput. Syst.*, 2014, pp. 2 :1–2 :70.
10. Klein (G.), Elphinstone (K.), Heiser (G.), Andronick (J.), Cock (D.), Derrin (P.), Elkaduwe (D.), Engelhardt (K.), Kolanski (R.), Norrish (M.) et al. – seL4 : Formal verification of an OS kernel. – In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pp. 207–220, 2009.
11. Kocher (P.), Genkin (D.), Gruss (D.), Haas (W.), Hamburg (M.), Lipp (M.), Mangard (S.), Prescher (T.), Schwarz (M.) et Yarom (Y.). – Spectre attacks : Exploiting speculative execution. *arXiv preprint arXiv :1801.01203*, 2018.
12. Leinenbach (D.) et Santen (T.). – Verifying the Microsoft Hyper-V hypervisor with VCC. – In Cavalcanti (A.) et Dams (D. R.) (édité par), *FM 2009 : Formal Methods*, 2009.
13. Liedtke (J.). – On micro-kernel construction. – In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, *SOSP*, SOSP, 1995.
14. Lipp (M.), Schwarz (M.), Gruss (D.), Prescher (T.), Haas (W.), Mangard (S.), Kocher (P.), Genkin (D.), Yarom (Y.) et Hamburg (M.). – Meltdown. *arXiv preprint arXiv :1801.01207*,

2018.
15. Moskal (M.), Santen (T.) et Schulte (W.). – VCC : A practical system for verifying concurrent C. – In *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009*, 2009.
16. Nipkow (T.), Wenzel (M.) et Paulson (L. C.). – *Isabelle/HOL : A Proof Assistant for Higher-order Logic*. – Berlin, Heidelberg, Springer-Verlag, 2002.
17. Parnas (D. L.) et Clements (P. C.). – A rational design process : How and why to fake it. *IEEE transactions on software engineering*, no2, 1986, pp. 251–257.
18. Richards (R. J.). – Modeling and security analysis of a commercial real-time operating system kernel. *In : Design and Verification of Microprocessor Systems for High-Assurance Applications*, pp. 301–322. – Springer, 2010.
19. Rushby (J.). – The design and verification of secure systems. – In *Eighth ACM Symposium on Operating System Principles (SOSP)*, pp. 12–21, 1981. – (ACM *Operating Systems Review*, Vol. 15, No. 5).
20. Rushby (J.). – A trusted computing base for embedded systems. – In *Proceedings 7th DoD/NBS Computer Security Initiative Conference*, pp. 294–311, 1984.
21. Rushby (J.). – *Partitioning in avionics architectures : Requirements, mechanisms, and assurance*. – Rapport technique, SRI International Menlo Park CA Computer Science Lab, 2000.
22. Samuel Hym and Veïs Oudjail. – `https://github.com/2xs/digger`.
23. Sanán (D.), Butterfield (A.) et Hinchey (M.). – Separation kernel verification : The xtratum case study. – In *Working Conference on Verified Software : Theories, Tools, and Experiments*, pp. 133–149. Springer, 2014.
24. Sekerinski (E.) et Sere (K.). – *Program development by refinement : case studies using the B method*. – Springer Science & Business Media, 2012, xxiiip.
25. Syeda (H.) et Klein (G.). – Reasoning about translation lookaside buffers. – In Eiter (T.) et Sands (D.) (édité par), *LPAR-21, 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Maun, Botswana, May 7-12, 2017*, *EPiC Series in Computing*, volume 46, pp. 490–508. EasyChair, 2017.
26. The Coq Development Team. – `https://coq.inria.fr`.
27. The Pip Development Team. – `https://github.com/2xs/pipcore`.
28. Wadler (P.). – Comprehending monads. – In *LISP and Functional Programming*, pp. 61–78, 1990.
29. Xu (F.), Fu (M.), Feng (X.), Zhang (X.), Zhang (H.) et Li (Z.). – A practical verification framework for preemptive OS kernels. – In *International Conference on Computer Aided Verification*. Springer, 2016.