Pas de Pannes, Pas d'Exploits: Vérification Automatique de Noyaux Embarqués

Olivier Nicole[1,2], Matthieu Lemerre[1], Sébastien Bardin[1], and Xavier Rival[2,3]

[1] *Université Paris-Saclay, CEA List, Saclay, France*
[2] *Département d'informatique de l'ENS, ENS, CNRS, PSL University, Paris, France*
[3] *Inria, Paris, France*

La sûreté et la sécurité de la plupart des ordinateurs dépendent de son composant le plus critique, le noyau, qui fournit les mécanismes de protections centraux. Les pires défauts de code d'un noyau sont :

— Des *erreurs à l'exécution*, qui conduisent le système entier à tomber en panne. Dans du code machine, il n'y a pas de comportement indéfini (comme en C), mais l'exécution d'une instruction qui conduit le code du noyau à lever une exception matérielle (e.g. opcode illégal, division par zéro, erreur de protection mémoire) est assimilable à une erreur à l'exécution ;
— Des *escalades de privilèges*, où un attaquant contourne les *auto-protections du noyau* et prend le contrôle du système entier. Dans le cas d'hyperviseurs, cela correspond au cas où l'attaquant parvient à s'échapper de sa machine virtuelle.

La seule manière de garantir qu'un noyau n'est pas sujet à ces erreurs est de le vérifier entièrement à l'aide de méthodes formelles [1]. Vérifier manuellement un noyau en utilisant un assistant de preuve [1–6] ou de la vérification déductive [7–10] peut garantir qu'un noyau satisfait une spécification formelle, permettant ainsi d'atteindre un haut niveau de sûreté et de sécurité. Mais cet effort est hors d'atteinte pour la plupart des acteurs du monde de l'embarqué, à la fois à cause de l'effort titanesque que demande l'écriture de milliers de lignes de preuves (e.g. 200,000 pour seL4 [1] ou 100,000 pour CertiKOS [11]) mais également à cause de la difficulté de trouver des experts à la fois en système et en méthodes formelles. Pour ces acteurs, la méthode idéale serait une vérification entièrement automatique, où les développeurs fourniraient seulement leur code et, avec très peu ou pas du tout de configuration, l'outil vérifierait automatiquement la propriété visée. De plus, une vérification complète devrait aller jusqu'au niveau de l'exécutable binaire, car 1. une large part du code d'un noyau de système embarqué consiste en des interactions bas niveau avec le matériel, et 2. la chaîne de compilation (options de compilation, compilateur, assembleur, éditeur de lien) peut introduire des bugs.

Les méthodes de vérifications de noyau récentes, appelées méthodes "presse-bouton" [12–14] sont basées sur l'exécution symbolique, ce qui a plusieurs avan-

tages : la méthode est suffisamment précise pour analyser du code machine sans se perdre, et la méthode marche de manière native avec des formules logiques et peut donc facilement être utilisée pour démontrer des propriétés de haut niveau spécifiées à la main. D'un autre côté, l'exécution symbolique en tant que technique de vérification souffre de limitations sévères, comme l'impossibilité de gérer les boucles non bornées, le besoin de fournir à la main les invariants du noyau (pour CertiKOS^S [13] : 438 lignes de spécifications pour 1845 instructions, i.e. un ratio de 23.7%) et une difficulté à passer à l'échelle à cause de l'explosion du nombre de chemins. Ces limitations sont inhérentes à l'exécution symbolique et ne peuvent pas être adressées sans un changement radical de méthode de vérification.

Notre but est de repousser les limites de ce que peut faire une méthode de vérification automatique de noyau, afin de la rendre pratique pour des développeurs systèmes. Nous nous concentrons sur les systèmes embarqués, qui sont caractérisés par une allocation mémoire plutôt statique, et par le fait qu'ils existent en général en de multiples variantes (selon le matériel ou l'application utilisant le noyau), ce qui rend l'automatisation de l'analyse très importante. Nos contributions sont les suivantes :

— Nous fournissons une méthode pour une vérification *entièrement automatique* de l'*absence d'escalade de privilège* et *l'absence d'erreurs à l'exécution* de petits noyaux de systèmes d'exploitation, *en contexte* (c'est-à-dire pour une disposition mémoire des applications donnée). Nous éliminons le besoin d'annotations manuelles : tout d'abord en développant un *interpréteur abstrait* [15] au niveau du code machine, qui permet d'analyser toute la *boucle système* (le code du noyau + une abstraction du code des applications utilisateur) pour *inférer* automatiquement les invariants du noyau (plutôt que de seulement les vérifier) ; deuxièmement en *prouvant* que l'absence d'escalade de privilèges est une propriété *implicite*, i.e. qui ne demande pas d'écrire une spécification (tout comme l'absence d'erreur à l'exécution [16]) ;
— Nous proposons une extension de la méthode pour la vérification *paramétrée* de noyau (i.e. la vérification du noyau indépendamment des applications). Les travaux précédents [12–14] modélisent la mémoire en utilisant un *modèle à plat* (la mémoire est un gros tableau d'octets, et les adresses sont représentées numériquement) qui empêche la vérification paramétrée et limite le passage à l'échelle et la précision [17] de la vérification. Nous proposons une représentation de la mémoire basée sur les types qui résout ces problèmes. Finalement, nous différencions le traitement du code d'initialisation (dont le but est d'établir les invariants du système) du traitement du code pendant l'exécution (dont le but est de préserver les invariants du système), ce qui améliore davantage la précision. Si notre méthode requiert un très petit nombre d'annotations manuelles, celles-ci remplacent la *précondition* sur l'application dont on aurait eu besoin sinon ;
— Nous avons conduit une évaluation approfondie sur deux études de cas, où nous démontrons que 1. il est possible d'implémenter un interpréteur abstrait sur du code machine qui soit suffisamment précis pour vérifier un noyau de

système d'exploitation industriel existant, sans modification, et avec très peu d'annotations (ratio $< 1\%$) ; 2. l'interprétation abstraite est utile comme un outil d'intégration continue pour détecter des défauts pendant le développement du noyau, surtout dans le cas de systèmes embarqués qui possèdent beaucoup de variantes ; 3. l'analyse paramétrée peut passer à l'échelle pour un grand nombre de tâches utilisateurs, tandis que notre analyse "en contexte" ne le peut pas.

Au final, nous adressons d'importantes limitations dans les méthodes automatiques existantes : nous pouvons faire une vérification *paramétrée* (indépendante des applications) ; nous gérons les boucles non bornées, nécessaires notamment pour implémenter des ordonnanceurs temps-réel ; et nous *inférons* les invariants du noyau, au lieu de seulement les vérifier. Comme dans la vérification formelle de noyau, *"le raisonnement sur les invariants domine l'effort de preuve"* [2] (dans seL4, 80% de l'effort fut passé à énoncer et vérifier des invariants [1]), ce travail est une étape-clé pour des vérifications automatiques de systèmes plus complexes.

Références

[1] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "seL4 : Formal verification of an OS kernel," in *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, ACM, 2009.

[2] B. J. Walker, R. A. Kemmerer, and G. J. Popek, "Specification and verification of the UCLA Unix security kernel," *Commun. ACM*, vol. 23, pp. 118–131, feb 1980.

[3] W. Bevier, "Kit : A study in operating system verification," *IEEE Transactions on Software Engineering*, vol. 15, pp. 1382–1396, 11 1989.

[4] R. J. Richards, *Modeling and Security Analysis of a Commercial Real-Time Operating System Kernel*, pp. 301–322. Boston, MA : Springer US, 2010.

[5] R. Gu, J. Koenig, T. Ramananandro, Z. Shao, X. N. Wu, S.-C. Weng, H. Zhang, and Y. Guo, "Deep specifications and certified abstraction layers," in *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, ACM, 2015.

[6] F. Xu, M. Fu, X. Feng, X. Zhang, H. Zhang, and Z. Li, "A practical verification framework for preemptive OS kernels," in *International Conference on Computer Aided Verification*, Springer, 2016.

[7] E. Alkassar, M. A. Hillebrand, W. Paul, and E. Petrova, "Automated verification of a small hypervisor," in *Verified Software : Theories, Tools, Experiments* (G. T. Leavens, P. O'Hearn, and S. K. Rajamani, eds.), (Berlin, Heidelberg), pp. 40–54, Springer Berlin Heidelberg, 2010.

[8] J. Yang and C. Hawblitzel, "Safe to the last instruction : automated verification of a type-safe operating system," *ACM Sigplan Notices*, vol. 45, no. 6, pp. 99–110, 2010.

[9] A. Vasudevan, S. Chaki, P. Maniatis, L. Jia, and A. Datta, "überSpark : Enforcing verifiable object abstractions for automated compositional security analysis of a hypervisor," in *25th USENIX Security Symposium (USENIX Security 16)*, USENIX Association, 2016.

[10] A. Ferraiuolo, A. Baumann, C. Hawblitzel, and B. Parno, "Komodo : Using verification to disentangle secure-enclave hardware from software," in *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, ACM, 2017.

[11] R. Gu, Z. Shao, H. Chen, X. N. Wu, J. Kim, V. Sjöberg, and D. Costanzo, "CertiKOS : An extensible architecture for building certified concurrent OS kernels," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, USENIX Association, 2016.

[12] M. Dam, R. Guanciale, and H. Nemati, "Machine code verification of a tiny ARM hypervisor," in *Proceedings of the 3rd International Workshop on Trustworthy Embedded Devices*, TrustED '13, ACM, 2013.

[13] L. Nelson, J. Bornholt, R. Gu, A. Baumann, E. Torlak, and X. Wang, "Scaling symbolic evaluation for automated verification of systems code with Serval," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, (New York, NY, USA), p. 225–242, Association for Computing Machinery, 2019.

[14] J. Nordholz, "Design of a symbolically executable embedded hypervisor," in *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, (New York, NY, USA), Association for Computing Machinery, 2020.

[15] P. Cousot and R. Cousot, "Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ACM, 1977.

[16] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival, "A static analyzer for large safety-critical software," in *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI'03)*, ACM, 2003.

[17] T. Reps, J. Lim, A. Thakur, G. Balakrishnan, and A. Lal, "There's plenty of room at the bottom : Analyzing and verifying machine code," in *International Conference on Computer Aided Verification*, Springer, 2010.