Programmation d'un noyau sécurisé en Ada

Arnauld Michelizza
arnauld.michelizza@ssi.gouv.fr

ANSSI

Résumé Alors que la majorité des vulnérabilités critiques des noyaux trouvent leur origine dans des débordements en mémoire ou l'usage incorrect de pointeurs, l'utilisation de langages sûrs pour l'implémentation de noyaux n'a été que rarement étudiée. L'article montre la faisabilité du développement d'un noyau en Ada et met en évidence la façon dont ce noyau est protégé des vulnérabilités les plus courantes.

1 Introduction

Plus de 50 ans après les débuts de l'informatique, bien peu de logiciels sont dépourvus de bugs et les noyaux des systèmes d'exploitation n'échappent pas à cette règle. Si un certain nombre de ces bugs ne se traduit que par des dysfonctionnements mineurs [6], d'autres sont particulièrement critiques car ils sont la cause de vulnérabilités exploitables à des fins malveillantes pouvant conduire à la corruption de tout le système.

Ceci est notamment le cas des systèmes d'exploitation usuels, dont le noyau est de conception monolithique. Dans ce type de noyau, n'importe quelle ligne de code du noyau peut accéder à l'ensemble de son espace d'adressage. Tout comme dans un programme classique, en n'importe quel point du noyau il est possible d'accéder à l'ensemble des variables et des structures globales. N'importe quelle erreur de programmation dans un service annexe peut ainsi faire planter l'ensemble du système, ou pire, permettre à un attaquant son exploitation pour obtenir un gain de privilèges ou l'accès à des informations confidentielles [8,9].

Différentes stratégies existent pour faire face à ce type de menaces :

- l'utilisation de contre-mesures *ad hoc* (canaris, zone de données non exécutable, etc.) ;
- l'adoption d'une architecture modulaire qui permet d'isoler les différents composants du noyau (Minix, Hurd) ;
- l'adoption de méthodes formelles pour prouver le respect de certaines propriétés par le noyau, comme par exemple l'absence de débordement de tableaux ou le non déréférencement de pointeurs nuls (SeL4).

Nous verrons qu'aucune solution n'est infaillible et que chacune a des avantages et des inconvénients. Loin d'être incompatibles, elles sont également complémentaires.

2 Renforcer la sécurité des systèmes d'exploitation : approches classiques

2.1 Adopter des contre-mesures *ad hoc*

L'utilisation de contre-mesures *ad hoc* s'inscrit dans une logique de défense en profondeur. À l'origine conçues pour durcir les programmes s'exécutant en espace utilisateur, certaines d'entre elles sont utilisées pour durcir le noyau [4].

Un article de Van Der Veen et *al.* [36] dresse un panorama synthétique des différentes techniques et contre-mesures liées aux vulnérabilités en espace utilisateur. Historiquement, l'une des premières contre-mesures a été d'empêcher l'exécution de code arbitraire injecté dans la pile [12], technique par la suite généralisée à toutes les zones de données [35].

Linux bénéficie de cette fonctionnalité grâce au *patch* Pax (KERNEXEC) [30] qui interdit l'exécution de code hors de la zone de texte du noyau sur les architectures x86, initialement grâce au support de la segmentation.

Les techniques d'exploitation classiques nécessitent de connaître les adresses mémoire utilisées. Une contre-mesure fut donc d'introduire de l'aléa dans l'emplacement des segments pour empêcher l'attaquant de les deviner [35]. Cette technique, appelée *Address Space Layout Randomization* (ASLR), a été implémentée partiellement au niveau du noyau Linux par PaX (RANDKSTACK) et de façon plus complète au niveau du noyau Windows.

Une autre contre-mesure a été l'introduction de canaris [10], des valeurs aléatoires injectées dans la pile ou dans le tas pour détecter les débordements de tableau. L'écrasement du canari par un débordement permet de détecter ce dernier avant le retour de la fonction, et par conséquent d'éviter l'exécution de code malveillant. Cette technique est implémentée au sein des noyaux de différents systèmes d'exploitation [4].

Le déréférencement de pointeurs en espace utilisateur est une source de vulnérabilité classique dans le noyau Linux. Comme le langage C n'offre pas de typage fort, le programmeur doit explicitement vérifier la validité des données transmises par l'utilisateur, typiquement à l'aide des fonctions `access_ok()`, `put_user()`, `get_user()`, `copy_from_user()` et

`copy_to_user()`. Le *patch* PaX (UDEREF) permet d'empêcher le déréférencement accidentel de ces pointeurs par le noyau, là encore initialement grâce à la segmentation [29].

Les apports de PaX dans le durcissement du noyau Linux sont notables, et il est important de souligner qu'à côté de nombreuses contre-mesures *ad hoc*, la sécurité de PaX est fondée sur l'utilisation de la segmentation pour distinguer clairement les différents espaces d'adressage (ce pour quoi elle était d'ailleurs originellement prévue [15]). De fait, le retrait de la segmentation dans les architectures *amd64* [34] a été du point de vue de la sécurité une régression, compensée partiellement par l'introduction plus tardive des bits SMAP (*Supervisor Mode Access Prevention*) et SMEP (*Supervisor Mode Execution Prevention*) sur les plateformes Intel64 [17].

Si l'ensemble de ces mesures est efficace pour protéger le noyau, elles ne protègent pas de tout. Leur application parfois incomplète et la sophistication de certaines attaques permettent de détourner le flot d'exécution du noyau via des techniques du type *return-to-libc* ou *ROP*. Enfin, rien n'interdit à un module compromis ou bogué d'accéder de façon arbitraire à la mémoire du noyau ou d'exécuter n'importe laquelle de ses fonctions.

2.2 Adopter une architecture de type micro-noyau

Tous les services du noyau n'ont pas besoin d'accéder à tout l'espace d'adressage du noyau. Une réponse à ce problème est donc d'exclure du noyau les modules non fondamentaux et de confiner leur contexte d'exécution afin qu'ils ne puissent accéder qu'aux données qui leur sont nécessaires. Cette approche met en œuvre trois principes de sécurité essentiels :

- le principe de séparation et de réduction de privilèges, chaque module ne peut accéder qu'à ses propres données et ne dispose pas des privilèges du superviseur ;
- le principe du *KISS* (*Keep It Simple and Stupid*), un noyau contenant moins de fonctionnalités est plus simple et donc plus facile à maintenir et à déboguer ;
- le principe de réduction de la surface d'attaque, un code plus petit voit sa surface d'attaque réduite.

Cette approche date du début de années 1970 qui a vu émerger les concepts de *security kernel* [28] et de *Trusted Computing Base* (TCB) [26]. Les efforts de l'époque étaient liés à l'émergence de systèmes multi-

utilisateurs à temps partagé et de systèmes multi-niveaux devant garantir l'absence de fuite d'information.

Ce type de conception pose certaines difficultés d'implémentation car le remplacement des appels de fonctions par un système d'IPC est complexe et pose des problèmes de performance spécifiques [20].

L'approche micro-noyau permet de confiner l'impact immédiat de la compromission d'un module. Toutefois, si cette approche apporte des résultats en termes de sûreté et de résilience [33], il n'est pas certain qu'elle traite le problème lié à la sécurité dans le sens où un service compromis, même en dehors du micro-noyau, peut affecter l'ensemble du système. Par exemple, si un attaquant parvient à corrompre ou à contrôler une pile TCP/IP implémentée en dehors du micro-noyau, l'injection ou la récupération de données affecte la sécurité de tout le système.

Par ailleurs, les micro-noyaux délèguent aux pilotes de périphériques l'accès aux registres matériels du périphérique en question, par exemple via des accès en MMIO [1]. Or certains registres permettent de contrôler l'accès au DMA, ce qui en l'absence d'IOMMU affecte là encore la sécurité de tout le système [7,26,27].

2.3 Prouver formellement le respect de certaines propriétés

L'approche formelle permet de prouver la correction de la conception et de l'implémentation de code. Elle est complémentaire de la conception micro-noyau et de l'approche par réduction de la TCB. Cette approche a été adoptée avec succès par le noyau seL4. Les 200 000 lignes de preuve (ISABELLE) ont permis de garantir la conformité d'environ 8 500 lignes de C avec la spécification abstraite du noyau [19].

S'il est possible, sous certaines conditions, de prouver un code restreint aux fonctionnalités limitées, il est difficilement envisageable de vérifier l'ensemble du code d'un noyau monolithique.

En premier lieu, prouver du code est difficile à mettre en œuvre, surtout quand il s'agit de code en langage C (11 ans-homme de travail pour 8500 lignes de C dans le cas de seL4).

Enfin, prouver le fonctionnement d'un noyau nécessite aussi de modéliser le comportement de la partie matérielle, ce qui peut s'avérer difficile, par exemple dans le cas d'une architecture *amd64*. Bien que SeL4 s'exécute sur un processeur plus simple (ARMv6), la bonne utilisation de la MMU n'a pas été prouvée car celle-ci n'a pas été modélisée [27].

1. *Memory-mapped I/O.*

Cette approche peut être vue comme étant l'approche idéale, mais les difficultés pour la mettre en œuvre la réservent à des projets limités en taille. L'approche telle que réalisée pour seL4, si elle offre une grande confiance, souffre d'être difficilement généralisable.

3 Supprimer les vulnérabilités par le choix d'un langage sûr

Si la plupart des vulnérabilités des systèmes d'exploitation peuvent être jugulées par des techniques d'ingénierie *ad hoc* associées à une conception modulaire, ces approches ne traitent pas la racine du problème, à savoir l'existence de défauts dans l'implémentation. L'approche formaliste semble idéale mais l'avancement actuel des techniques dans ce domaine restreint son cadre d'application. Afin d'identifier d'autres axes d'approche, nous nous sommes interrogé sur l'origine principale des vulnérabilités des noyaux.

3.1 Analyse des principales sources de vulnérabilités des noyaux

La recherche menée par Chen et al. [6] étudie les sources de vulnérabilité du noyau Linux sur une période allant de 2010 à 2011. Les vulnérabilités les plus critiques permettent d'écrire à un endroit arbitraire du noyau ou de la mémoire. Elles sont causées essentiellement par des *buffer overflows* (41%), des *integer overflows* (37%) et des problèmes de pointeurs adressant des zones invalides (19%).

Pour confirmer cette étude, nous avons procédé à notre propre analyse des vulnérabilités du noyau Linux sur une période d'environ un an à partir de 2011 et en nous appuyant sur les données de la base CVE du MITRE [11]. Nous n'avons retenu que les vulnérabilités les plus critiques, c'est-à-dire celles conduisant à une escalade de privilèges ou à l'obtention d'accès mémoire arbitraires.

	Buffer overflow	*Integer overflow*	Typage	Pointeur	Conception
Total	9	2	2	1	3

Figure 1. Types d'erreurs à l'origine de 17 vulnérabilités critiques

Sur les 17 vulnérabilités retenues, 13 (soient 76%) sont liées à des débordements en mémoire (*buffer-*, *heap-*, *stack-*, *integer overflow*) ou à des problèmes de typage divers, et une seule est due à un pointeur invalide

(*use after free*). Les 3 vulnérabilités restantes sont liées à des erreurs plus générales de conception.

Ce résultat concorde avec celui de Chen et *al.* qui identifie les débordements comme cause principale (78%) des vulnérabilités donnant lieu à des accès arbitraires en mémoire. Utiliser un langage garantissant l'absence de débordements semble donc être une alternative pour réduire le nombre de vulnérabilités des systèmes d'exploitation.

3.2 Problèmes du langage C

Le langage C a été conçu à l'origine comme remplacement de l'assembleur pour écrire les premiers systèmes Unix [25]. De fait, le C offre la possibilité de gérer explicitement les allocations de mémoire, de contrôler finement la représentation bas-niveau des données manipulées et de produire un code qui s'exécute rapidement. Ces caractéristiques sont indispensables à la programmation système, et par conséquent, le C est devenu le langage le plus populaire pour la programmation de noyau dès les années 1970.

En pratique, le C est un langage permettant d'accéder à la mémoire brute sans se soucier du type des objets manipulés. L'accent est surtout mis sur les opérations qui affectent les objets plus que sur le type des objets eux-mêmes car le C, en tant que successeur du BCPL et du B, est dans son essence un langage non typé [25]. Le typage en C est par conséquent assez peu contraignant, et aucune vérification n'est faite à l'exécution.

Tout langage de programmation offre un ensemble d'abstractions de la machine sur laquelle le programme s'exécute, comme les instructions du processeur et la mémoire, mais aussi de certains mécanismes, comme les appels de fonctions, les variables et les types.

Plus précisément, un type est une annotation qui permet de définir certaines propriétés possédées par un élément. Par exemple, en C, le type tableau est une abstraction permettant de définir un objet occupant une portion de mémoire. Dans l'exemple suivant, un tableau de `char` pouvant contenir 5 éléments est déclaré :

```
char tab[5];
```

Ce genre de déclaration est courant. Le problème est que le C permet facilement de contourner cette abstraction :

```
char tab[5];
tab[5] = 'a' ; /* buffer overflow */
```

L'erreur est ici évidente, elle passe toutefois inaperçue du compilateur gcc qui ne produit aucun message d'avertissement même en utilisant les paramètres `-Wall -W`.

Bien que le C ait une sémantique et une syntaxe pour définir un tableau de taille fixe, il n'offre pas de garantie quant au maintien de cette abstraction à l'exécution, c'est par conséquent au développeur d'assurer son maintien. Or la faiblesse du typage du C ne protège pas contre des erreurs de programmation parfois subtiles qui sont à l'origine de vulnérabilités classiques dans ce langage, telles que les dépassements de tableau (*array out-of-bounds access*), l'usage de pointeurs invalides (*dangling pointer*, *double free* et *use after free*) et les erreurs de conversion de type.

3.3 Choisir un langage de programmation système sûr

Cette section traite des recherches ayant étudié le remplacement du C ou du C++ par un langage sûr (*safe language*), afin de supprimer les vulnérabilités liées aux pointeurs invalides et aux débordements.

Pierce, dans *Types and programming languages* [24], définit la notion de sûreté (*safety*) rapportée aux langages de programmation de cette façon :

> *Safe language is one that protects its own abstractions. [...] safety refers to the language's ability to guarantee the integrity of these abstractions. [...] A programmer [...] expects that an array can be changed only by using the update operation on it explicitly – and not, for example, by writing past the end of some other data structure. [...] Language safety can be achieved by static checking, but also by run-time checks that trap nonsensical operations just at the moment when they are attempted and stop the program or raise an exception.*

Autrement dit, un langage sûr (*safe language*) doit permettre de détecter toutes les erreurs affectant les types lors de la compilation, grâce à l'analyse statique, ou bien doit être capable de les traiter correctement et proprement, lors de l'exécution. La règle d'or est qu'aucune erreur ne doit pouvoir mener à un comportement imprédictible du système.

L'utilisation de languages tels que Java, Haskell ou OCaml permet d'éviter la plupart des violations de type. Cependant, ces langages ne sont pas adaptés à la programmation bas niveau car ils n'offrent pas au programmeur le contrôle précis sur la représentation machine des données et ils ne permettent pas de gérer manuellement la mémoire.

Un langage sûr ne peut maintenir sa cohérence qu'en empêchant les accès directs aux données. De ce fait, dans la plupart des langages de

programmation, les questions relatives à la représentation des données sont distinctes de la sémantique du langage. Or, ceux-ci sont parfois nécessaires et notamment en programmation système où la maîtrise de la représentation matérielle des données est essentielle. En C, la flexibilité pour représenter et manipuler les données se fait aux dépens du respect du typage. Peng Li [23] insiste donc sur le compromis nécessaire entre la sûreté du langage, qui impose des contraintes afin de maintenir la cohérence de ses abstractions, et la flexibilité, qui permet de répondre aux besoins de la programmation bas-niveau.

Les compilateurs sont capables de détecter certaines erreurs lors de la compilation, cependant, toutes ne sont pas détectables par l'analyse statique. Dans certains cas, ces erreurs ne peuvent être détectées que lors de l'exécution du programme. Par conséquent, certains chercheurs ont proposé d'étendre le langage C afin d'y inclure des directives particulières ou des types nouveaux qui permettent de vérifier la cohérence du typage à l'exécution. L'article de Peng Li passe en revue un ensemble de langages de recherche, tous dérivés du C (CCured [21], SafeC [22], Cyclone [18] et Vault [13]) qui proposent de combiner analyse statique et vérifications à l'exécution. Quand l'analyse ne peut déterminer si un pointeur est utilisé de façon sûre, des vérifications à l'exécution garantissent son utilisation sans danger.

Plutôt que d'étendre le langage C, d'autres chercheurs ont choisi de créer un nouveau langage conciliant sûreté du typage et suffisament de flexibilité pour développer un système d'exploitation. Ceci est par exemple le cas du langage fonctionnel BitC [31], créé par Shapiro pour la programmation de systèmes sûrs.

Cette approche par le langage permet de traiter la cause des failles d'implémentation les plus courantes. Toutefois, les recherches évoquées partagent toutes le même inconvénient. Elles proposent un nouveau dialecte ou un nouveau langage, créé spécialement pour l'occasion et utilisable uniquement à des fins de recherche. Aucun d'eux n'est plus maintenu, ce qui rend difficile leur utilisation au sein de projets industriels.

Curieusement, le langage Ada n'est évoqué dans aucun de ces articles, excepté celui de Shapiro sur l'origine du langageBitC [31]. S'il considère que le langage Ada et son dérivé Spark [3,5] sont probablement les langages et les outils actuels les plus adaptés pour ce type de projet, il leur reproche de ne pas prendre en compte des avancées de la théorie des langages de programmation qui ont été faites ces vingt dernières années, sans toutefois préciser quels sont ces manques.

3.4 Le langage Ada

Dans les années 1970, le département de la défense des États-Unis (DoD) créa un groupe de travail afin de choisir un langage de programmation adapté à ses projets. Ada, conçu par Jean Ichbiah (Honeywell Bull), remporta la sélection en réponse au cahier des charges rédigé par le DoD. Normalisé en 1983, le langage a évolué pour donner plusieurs révisions : Ada95, Ada 2005 [2], et plus récemment Ada 2012 [1]. Ce langage est aujourd'hui supporté par la société AdaCore qui maintient et fait évoluer Gnat, le portage de Ada pour GNU/gcc.

Le langage Ada possède deux propriétés essentielles. La première est son typage fort qui garantit que la plupart des erreurs sont détectées à la compilation ou à l'exécution. La seconde est qu'il a été conçu dès l'origine pour permettre la programmation bas-niveau, et notamment dans l'embarqué. Ada est ainsi utilisé dans l'aéronautique, l'aérospatial, le ferroviaire et en général dans tous les domaines où la sûreté de fonctionnement est critique [14]. Ces propriétés ainsi que les autres aspects du langage sont détaillées plus loin.

4 Développement d'un noyau en Ada

Si les propriétés du langage Ada en font un langage adapté à la programmation bas-niveau, nous avons voulu voir s'il était possible de faire un noyau de système d'exploitation en Ada. Plus précisément, nous avons souhaité voir si les mécanismes de vérification à l'exécution, qui fonctionnent pour des applications en espace utilisateur, sont transposables à un noyau. Nous avons également voulu mesurer les performance d'un noyau en Ada comparé à son équivalent en C. Ce dernier point est important car des temps d'exécution trop lents peuvent remettre en question la pertinence de ce langage pour des développements industriels.

Alors que le langage Ada présente de nombreuses qualités, il est surprenant de constater sa faible utilisation au sein de projets informatiques en dehors des domaines où la sûreté de fonctionnement est critique. Nous avons donc fait l'expérience de ce langage que nous ne connaissions pas avant cette étude.

Finalement nous sommes parvenu assez rapidement à développer un noyau rudimentaire pour plateforme *amd64* [17,34] et s'exécutant en *long mode* (64 bits). Ce noyau fait environ 11 000 lignes de Ada et un peu moins de 400 lignes d'assembleur. Il gère la mémoire segmentée (autant que le processeur le permette dans ce mode) et paginée. Il est multitâches préemptif, avec un support partiel (en lecture seule) du système

de fichiers ext2 qui permet de lire des fichiers et de lancer des exécutables au format ELF-64 en espace utilisateur. Il contient des pilotes permettant d'accéder au disque dur (de type *ATA*) en mode PIO ou à la carte réseau (*Intel Pro 1000/MT*). Enfin, une pile IPv6 permet au noyau de répondre aux requètes ICMPv6.

4.1 Facilités et difficultés rencontrées

Notre prise en main de Ada a été difficile. Nous avions l'habitude et les réflexes du C, or développer en Ada nécessite de penser autrement la conception logicielle pour intégrer la logique propre à ce langage.

Par exemple, lors de la conception d'un noyau, une étape assez primitive est le développement de fonctions pour afficher des messages à l'écran via l'écriture directe en mémoire vidéo. En C, une méthode pour écrire dans cette mémoire est de calculer le déplacement par rapport à sa base (projetée en RAM à l'adresse 0xb8000), et d'affecter directement les valeurs qui codent le caractère et ses attributs (couleur, fond, clignotement, etc.) :

```
volatile char *video = (volatile char *) (0xb8000 + 2 * column + 160 *
    line);
*video = c;
*(video + 1) = color;
```

Les contraintes du langage Ada concernant l'arithmétique des pointeurs et le respect du typage interdisent ce genre d'écriture. Tout élément manipulé doit avoir un type clairement défini. Dans cet exemple, il faut d'abord définir un type composite pour chaque caractère en mémoire vidéo, puis il faut ensuite définir un type pour la mémoire vidéo elle-même (ici sous forme d'un tableau à deux dimensions) :

```
-- character
type t_char is
   record
      char : unsigned_8;
      attr : unsigned_8;
   end record;
pragma pack (t_char);

-- screen
subtype t_column is integer range 1 .. 80;
subtype t_line   is integer range 1 .. 25;

type t_video_ram is array (t_line, t_column) of t_char;
pragma pack (t_video_ram);

video : t_video_ram;
for video'address use address (16#b8000#);
```

La clause `'address` indique au compilateur l'emplacement en mémoire d'un objet. Elle est utilisée ici pour préciser l'emplacement du tableau de caractères correspondant à la mémoire vidéo. L'affichage d'un caractère à l'écran peut alors se faire comme ceci :

```
video(line, column).char := to_unsigned_8 (c);
video(line, column).attr := color;
```

Cet exemple illustre une différence essentielle entre le langage C et Ada. Là où le C peut être vu comme un macro assembleur permettant de manipuler directement le matériel, le langage Ada oblige à définir des abstractions.

Une autre difficulté du langage Ada est qu'il a été conçu dès son origine pour pouvoir régler des problèmes assez différents. Par conséquent, il possède de nombreux aspects et de nombreuses fonctionnalités qui le rendent difficile à appréhender. Il nous a personnellement fallu plusieurs mois pour nous sentir à l'aise avec ce langage. De fait, la dernière édition du manuel de référence de Ada [1] contient plus de 800 pages [2].

Passée la phase de développement, la mise au point et le débogage sont particulièrement facilités par le mécanisme d'exceptions. En C une erreur sur un type (par exemple un *buffer overflow*) peut se révéler assez tard à l'exécution par rapport au moment où elle intervient, et de fait être la source de bugs difficiles à corriger. En Ada, le mécanisme de vérifications à l'exécution génère, en cas d'erreur, une exception qui indique précisément le fichier, la ligne et la cause de l'erreur. Nous détaillerons ce mécanisme plus loin.

4.2 Empreinte de Ada à l'exécution

Ada contient des fonctionnalités qui se traduisent par une empreinte à l'exécution. Il est tout à fait possible de n'utiliser qu'une partie de ces fonctionnalités, et donc de réduire cette empreinte.

Par exemple, Ada supporte nativement la programmation concurrente grâce à un mécanisme de gestions de tâches et de passage de messages. Cette fonctionnalité n'a pas été retenue car, outre les difficultés pour l'implémenter, son intérèt ne nous a pas semblé évident pour ce projet.

2. Pour ces raisons, Tony Hoare a critiqué de manière virulente le langage Ada [16]. Toutefois, cette critique date de 1980 alors que le langage n'a été standardisé qu'en 1983, période au cours de laquelle le langage a été simplifié. De fait, Hoare a mis en avant les qualités de Ada au sein d'une préface en 1987 [37].

Une autre fonctionnalité concerne la possibilité pour une fonction de retourner un tableau de taille indéfinie. Cette fonctionnalité est pratique, mais elle nécessite l'allocation d'une seconde pile noyau pour chaque processus utilisateur et la réimplémentation du package standard Ada `System.Secondary_Stack`. Outre son implémentation difficile, le prix à payer est ici une empreinte mémoire que nous avons jugé trop importante.

La seule empreinte spécifique à Ada conservée par notre noyau concerne l'utilisation des vérifications dynamiques de typage. Ces vérifications à l'exécution sont associées à un mécanisme de gestion des exceptions, dont le but est de traiter les erreurs détectées. L'ensemble de ce mécanisme est détaillé plus loin. Notons que toutes les fonctionnalités de ce mécanisme n'ont pas été retenues, comme la possibilité de définir de nouvelles exceptions ou d'insérer une gestion contextuelle des exceptions au sein du code. Outre les difficultés d'implémentation, le bénéfice de ces fonctionnalités nous semblait discutable.

4.3 Vérifications des erreurs de typage et gestion des exceptions

Le compilateur Ada ne peut pas toujours prendre de décision concernant la correction du typage grâce à l'analyse statique, lors de la compilation. Dans ce cas, il ajoute des instructions assembleur au code afin de vérifier à l'exécution la correction du typage. Considérons par exemple le type `small` défini ci-dessous et dont les valeurs sont bornées entre 1 et 10 :

```
type small is new integer range 1 .. 10;
```

Lors des opérations sur des données de ce type, comme par exemple une incrémentation ou une décrémentation, le compilateur va ajouter des instructions assembleur pour vérifier l'absence de débordement. En cas d'erreur, une exception est déclenchée.

Par défaut, les exceptions sont gérées par un ensemble de procédures qui interrompent le programme et affichent un message d'erreur indiquant sa cause et où elle se situe précisément. Par exemple, si une variable de type `small` vaut 10, tenter de lui ajouter 1 génère cette exception :

```
raised CONSTRAINT_ERROR : main.adb:9 range check failed
```

Les exceptions peuvent aussi être traitées par un gestionnaire d'exceptions local, défini par le développeur, comme dans cet exemple :

```
begin
   s := s + 1;
exception
   when constraint_error =>
      put ("failure");
      return;
end ;
```

Listing 1. Gestion locale d'exception

Gestion des exceptions dans le noyau Le compilateur Ada, par défaut, génère du code assembleur qui vérifie la validité du typage des variables manipulées et permet de traiter les erreurs quand elles surviennent. La procédure ci-dessous est susceptible de faire déborder la variable s, dont le type `small`, défini précédement, n'accepte que les valeurs entre 1 et 10 :

```
procedure inc (s : in out small) is
begin
   s := s + 1;
end inc;
```

Listing 2. Incrémentation d'une variable

Nous avons compilé ce code avec l'option `-gnatVa` qui inclut l'ensemble des vérifications dynamiques possibles en Ada. Cette option n'est en principe utilisée que pour le débogage mais la criticité du code noyau justifie selon nous l'adoption d'un tel niveau de vérification. Le compilateur génère alors le code assembleur suivant :

```
foo__inc:
        leal    -1(%rdi), %eax
        subq    $8, %rsp
        cmpl    $9, %eax
        ja      .L5
        leal    1(%rdi), %eax
        cmpl    $11, %eax
        je      .L6
        addq    $8, %rsp
        ret
.L5:
        movl    $6, %esi
        movl    $.LC0, %edi
        call    __gnat_rcheck_06
.L6:
        movl    $6, %esi
        movl    $.LC0, %edi
        call    __gnat_rcheck_12
```

Listing 3. Code assembleur avec les vérifications de débordement

En cas d'erreur, une procédure permettant de traiter l'exception est appelée. Chaque procédure correspond à un type d'erreur et prend en argument le nom du fichier et le numéro de la ligne où l'erreur a été détectée. Il existe en tout 35 procédures distinctes pour autant de types d'erreurs différentes.

Le code précédent montre que les procédures `__gnat_rcheck_06` et `__gnat_rcheck_12` sont susceptibles d'être appelées. Elles correspondent respectivement aux erreurs `invalid data` et `range check failed`. Ces procédures sont définies et implémentées dans le paquetage standard `Ada.Exceptions`, que nous avons réécrit pour les besoins de notre noyau. Par exemple, la procédure `__gnat_rcheck_06` est implémentée de cette façon :

```
procedure Rcheck_06 (File : Types.Kernel_Address; Line : Integer) is
begin
   Alert_Msg (Rmsg_06, File, Line);
end Rcheck_06;
```

Listing 4. Procédure `Rcheck_06`

Cette procédure appelle `Alert_Msg` qui affiche un message d'erreur, contenant le fichier, la ligne et la nature de l'erreur à l'origine de l'exception, puis qui arrête le système.

En l'occurence, notre noyau traite toutes les exceptions de la même façon en affichant un message et en arrêtant le système. Or si l'erreur dans le code du noyau n'offre pas à l'attaquant de vulnérabilité exploitable, elle cause un déni de service qui peut avoir des conséquences importantes dans un environnement de production. Toutefois, cela permet de rendre l'attaque, mais aussi l'erreur de programmation, visibles. Notons qu'il aurait été possible de traiter autrement l'exception, par exemple en affichant un message d'erreur et en ne terminant que le processus en cours.

Chaque exception correspond à une erreur dans le code du noyau qu'il est nécessaire de corriger. Les renseignements précis sur la cause mais aussi sur la localisation de l'erreur dans le code facilitent l'amélioration de la qualité globale de d'implémentation.

4.4 Désactivation des vérifications

Le code Ada présenté à la section précédente et correspondant à une simple incrémentation (*listing* 2) génère un code assembleur volumineux

(*listing* 3) du fait des vérifications dynamiques. Il est possible de supprimer ces vérifications, en totalité ou partiellement, grâce à certaines directives :

```
procedure inc (s : in out small) is
   pragma suppress (range_check, s);
begin
   s := s + 1;
end inc;
```

Le code assembleur produit est beaucoup moins volumineux :

```
foo__inc:
        leal    1(%rdi), %eax
        ret
```

Listing 5. Code assembleur sans vérification de débordement

Nous voyons ici que le coût des vérifications est important, avec 15 instructions assembleur (dont 9 sont exécutées si tout va bien), au lieu de 2. Nous analysons plus loin, de façon plus exhaustive, le coût en mémoire mais aussi à l'exécution de cette empreinte des vérifications sur le code assembleur généré.

Toutefois, si la désactivation des vérifications est possible, elle est souvent dangereuse car ce sont ces vérifications qui garantissent la correction du typage. Ces désactivations ne devraient donc survenir que dans le cas où le fonctionnement est prouvé.

Spark Le langage Spark, basé sur Ada, fournit un ensemble d'annotations et d'outils de preuve permettant notamment de détecter par l'analyse statique les violations de typage et de prouver certaines propriétés. Spark est conçu pour le développement d'applications critiques, pour lesquelles le déclenchement d'exceptions à l'exécution n'est pas envisageable.

En prouvant formellement l'absence de problèmes de typages dans le code, Spark permet de désactiver l'ensemble des vérifications dynamiques insérées par défaut en Ada. Outre le gain en sûreté, il y a donc un gain en performance. Toutefois, l'utilisation de Spark pose aussi des contraintes, comme par exemple l'absence d'allocation dynamique, qui peuvent limiter son domaine d'application.

Nous avons utilisé Spark pour prouver l'absence de violation de typage dans les fonctions d'affichage, initiallement écrites en Ada. Ces vérifications sont réalisées automatiquement, sans besoin d'annotations particulières. Les annotations sont utilisées par les outils d'analyse pour

vérifier la correction de l'implémentation et la cohérence du flot de données. À titre d'exemple, le code ci-dessous montre la déclaration de la fonction `inc_cursor_line` qui incrémente le curseur d'une ligne vers le bas :

```
procedure inc_cursor_line;
--# global  in out cursor, screen;
--# derives cursor from cursor &
--#         screen from screen, cursor;
--# post cursor.column = cursor~.column
--#   and (cursor~.line < t_line'last ->
--#           cursor.line = cursor~.line + 1)
--#   and (cursor~.line = t_line'last ->
--#           cursor.line = cursor~.line);
```

Listing 6. Déclaration de la procédure `inc_cursor_line`

Les annotations indiquent ici la façon dont cette procédure modifie le curseur et l'écran. On y voit par exemple que l'état final du curseur dépend seulement de son état initial. Des annotations supplémentaires décrivent précisément comment la ligne et la colonne où se situe le curseur doivent être changées.

4.5 Sérialisation de données (*marshalling*)

La sérialisation et la désérialisation de données sont indispensables à l'implémentation des appels systèmes. Cette section décrit comment le langage Ada permet de désérialiser les données en provenance de l'espace utilisateur et leur applique les contraintes inhérentes au typage utilisé.

À titre d'exemple, considérons une variable de type `small` :

```
type small is new integer range 1 .. 10;
value : small;
```

Lors d'un appel système 64-bit, par convention, le premier argument transmis par l'utilisateur est encodé dans le registre RDI. La valeur de ce registre peut être directement récupérée dans `value` grâce à l'inclusion d'instructions en assembleur au sein du code Ada :

```
system.machine_code.asm
  ("mov %%rdi, %0",
   outputs  => small'asm_output ("=g", value),
   volatile => true);
```

Le problème est que le code assembleur généré ne contient aucune vérification du respect du typage de la valeur transmise :

```
mov %rdi, %eax
```

Il est toutefois possible de forcer la vérification en appliquant l'attribut `valid`. Le code suivant vérifie que la valeur transmise par l'utilisateur est bien du type `small` et déclenche un exception dans le cas contraire :

```
system.machine_code.asm
  ("mov %%rdi, %0",
   outputs  => small'asm_output ("=g", value),
   volatile => true);

if not value'valid then
   raise constraint_error;
end if;
```

L'observation du code assembleur généré montre l'inclusion des vérifications susceptibles de lever une exception en cas de valeur invalide :

```
.LC0:
    .ascii  "foo.adb"
    ...
    mov %rdi, %eax
    leal    -1(%rax), %edx
    cmpl    $9, %edx
    ja      .L4
    ...
.L4:
    movl    $14, %esi
    movl    $.LC0, %edi
    call    __gnat_rcheck_04
```

4.6 La sécurité du noyau en pratique

Cette section illustre par des exemples la sécurité apportée à notre noyau par le typage fort.

Protection contre les fuites d'information du noyau Le langage Ada contient par défaut un type `address`. Il s'agit d'un type modulaire non-signé qui contient certaines restrictions afin de préserver la sémantique apportée par cette abstraction. Par exemple, il est impossible en Ada d'additionner deux adresses car cette opération n'a pas de sens. En revanche, on peut y ajouter un offset (de type `storage_offset`) pour obtenir une nouvelle adresse.

Plutôt que d'utiliser directement le type `address` fourni par Ada, il nous a semblé plus judicieux de définir et d'utiliser des types spécifiques en fonction du type d'adresse manipulé par le noyau. Il

s'agit des types virtual_address, kernel_address, user_address et physical_address :

```
type physical_address is mod 2 ** 64;

type virtual_address is mod 2 ** 64;

type kernel_address is new virtual_address
   range 16#ffff_8000_0000_0000# .. 16#ffff_ffff_ffff_ffff#;
for kernel_address'size use 64;

type user_address is new virtual_address
   range 16#0000_0000_0000_0000# .. 16#0000_7fff_ffff_ffff#;
for user_address'size use 64;
```

L'utilisation de ces types rend le code plus clair en permettant de connaître à chaque fois le type exact d'adresse utilisé. Mais cela permet aussi d'éviter certains bugs causés par les conversions implicites, impossibles en Ada.

Les conversions sont souvent nécessaires car une même valeur peut être appréhendée différement selon le code qui la manipule. Or les conversions sont dangereuses car elles peuvent permettre de violer l'abstraction fournie par le typage.

Il existe plusieurs mécanismes de conversion, les conversions *vérifiées* [3] ne sont applicables qu'à des types compatibles entre eux, comme par exemple deux types numériques, alors que les conversions *non vérifiées* [4] sont utilisées pour les types sans rapport évident entre eux, comme par exemple un accès vers une adresse ou un entier vers un caractère.

Les conversions *non vérifiées* sont dangereuses car l'absence de vérification permet l'introduction de valeurs invalides. Elles nécessitent l'utilisation explicite de fonctions définies par le développeur et instanciées à partir de la fonction générique ada.unchecked_conversion. Le noyau les utilise par exemple, pour convertir un accès sur une table de page en une addresse du noyau :

```
function to_kernel_address is
   new ada.unchecked_conversion (t_ptable_access, kernel_address);
```

Pour revenir à la protection du noyau par le typage, prenons l'exemple d'un appel système qui affiche à l'écran une chaîne de caractères. Cet appel prend en paramètre l'adresse d'une chaîne de caractères située dans l'espace utilisateur :

3. Ada Reference Manual. *Section 4.6, Type Conversions.*
4. *Ibidem. Section 13.9, Unchecked Type Conversions.*

```
print ("hello, world\n");
```

Comme l'utilisateur contrôle la valeur transmise, il peut essayer de transmettre une adresse située dans l'espace du noyau, par exemple pour obtenir des informations exploitables pour monter une attaque ou bien accéder à des données confidentielles :

```
print ((char*) 0xffff800000000000);
```

Le gestionnaire d'appels systèmes, implémenté dans le paquetage `kernel.syscall`, récupère les paramètres transmis conformément à l'API *amd64*. Le premier argument, transmis par le registre RDI, est récupéré dans la variable `rdi`. La récupération de cette valeur utilise le mécanisme de désérialisation évoqué en section 4.5.

Dans l'exemple suivant, l'utilisation par le noyau d'une conversion *vérifiée*, dont le nom reprend celui du type cible (ici `user_address`), permet à celui-ci de s'assurer que l'argument transmis est bien une adresse utilisateur :

```
108 case syscall is
109    -- print
110    when 1 =>
111       declare
112          string_addr : constant user_address := user_address (rdi);
113       begin
114          syscall_print (string_addr);
115       end;
```

La tentative par l'utilisateur d'afficher le contenu de la mémoire en espace noyau lève une exception qui produit le message d'erreur suivant :

```

kernelsyscall.adb +112: range check failed
```

On remarque que le nom de fichier, le numéro de ligne et le type d'erreur correspondent exactement à l'endroit supposé d'une levée d'exception par la conversion.

Un développeur noyau optimiste pourrait passer outre ce mécanisme de vérification pour utiliser directement la valeur transmise par l'utilisateur :

```
108 case syscall is
109    -- print
110    when 1 =>
111       declare
112          string_addr : constant user_address;
```

```
113          for string_addr'address use rdi'address;
114       begin
115          syscall_print (string_addr);
116       end;
```

Cet exemple utilise une clause de représentation pour indiquer au compilateur que la variable `string_addr` est située au même endroit que la variable `rdi`, ce qui permet de faire l'économie d'une conversion vérifiée. Pourtant, ce code génère aussi une exception :

```
kernelsyscall.adb +115: invalid data
```

L'explication est que le compilateur génère du code assembleur pour vérifier la validité de la valeur transmise à la procédure `syscall_print` avant de l'appeler.

Tous ces exemples illustrent une différence essentielle entre le langage C et le langage Ada. Alors que le C ne fait aucun contrôle concernant le typage et permettra l'utilisation de valeurs invalides de façon totalement transparente, Ada va constamment vérifier le typage des variables pour détecter les erreurs et les traiter.

Pour reprendre l'exemple précédent, voici le code qui permet de vérifier la validité du paramètre transmis lors de l'appel système tout en gérant proprement les erreurs et sans forcement générer d'exception :

```
108 case syscall is
109    -- print
110    when 1 =>
111       declare
112          string_addr : constant user_address;
113          for string_addr'address use rdi'address;
114       begin
115          if string_addr'valid then
116             syscall_print (string_addr);
117          else
118             -- gestion propre de l'erreur
119             ...
120          end if;
```

L'utilisation de types spécifiques pour manipuler les différents espaces d'adressage apporte une sécurité importante : il est presque impossible de mélanger accidentellement données du noyau et données de l'espace utilisateur.

Protection contre les *buffer overlow* Le langage Ada dispose d'une sémantique qui permet de limiter considérablement le risque de *buffer*

overflow et plus particulièrement d'erreurs de types *off-by-one* lors d'accès à un tableau. Par exemple, là où en C on écrit :

```
char buf[1024];
int i;
for (i=0; i<1024; i++)
   ...
```

On écrirait en Ada :

```
for i in buf'range
loop
   ...
end loop;
```

Illustrer comment le noyau est protégé contre les *buffer overlow* n'est donc pas simple car le langage Ada invite naturellement à utiliser certains éléments sémantiques et syntaxiques qui tendent à immuniser le code contre ce type de danger. Nous avons par conséquent créé un exemple de toute pièce au sein du noyau :

```
136 declare
137    tab : array (1 .. 10) of integer;
138 begin
139    for i in 1 .. 11
140    loop
141       tab(i) := i;
142    end loop;
143 end;
```

Ce code tente d'affecter une valeur en dehors du tableau, ce qui déclenche une exception :

```
kernel.adb +141: index check failed
```

Notons au passage que le compilateur ne nous aurait pas autorisé à affecter directement une valeur en dehors du tableau :

```
139 tab(11) := 42;
```

Ce code génère une erreur à la compilation :

```
kernel.adb:139:05: warning: value not in range of subtype of "Standard.
    Integer" defined at line 3
kernel.adb:139:05: warning: "Constraint_Error" will be raised at run
    time
gnatmake: "kernel.adb" compilation error
```

Protection contre les pointeurs invalides Ada ne contient pas à proprement parler de type pointeur. L'équivalent Ada est le type *accès* qui ne peut pointer que sur un type précis et qui doit toujours avoir une valeur légitime (éventuellement `null`). Le déréférencement d'un accès est toujours précédé par une vérification de sa valeur. Un accès de valeur `null`, non initialisé ou dont l'objet a été désalloué, ne peut être déréférencé, ce qui protége le noyau des attaques classiques exploitant les pointeurs invalides (*dangling pointer*, *use after free*, etc.).

Le code suivant présente un exemple de déclaration et d'affectation d'un type accès vers un entier. Le type accès doit être défini et la variable accédée doit être explicitement marquée par la directive `aliased` :

```
type integer_access is access all integer;
i : aliased integer := 42;
p : integer_access := i'access;
```

Il est aussi possible d'allouer dynamiquement un objet grâce au mot réservé `new` qui appelle un allocateur (son implémentation au sein de notre noyau est abordée plus loin) :

```
p := new integer;
p.all := 42;
```

Ada n'incluant pas de *garbage collector*, les désallocations doivent être faites manuellement. Pour cela, la fonction de désallocation doit être instanciée explicitement à partir de la fonction générique `ada.unchecked_deallocation` :

```
procedure free_integer is
      new ada.unchecked_deallocation (integer, integer_access);
```

L'exemple suivant illustre la protection apportée par Ada lors de la manipulation d'accès. Ce code, qui tente d'allouer un entier après avoir libéré la mémoire, génère une exception :

```
10 p := new integer;
11 p.all := 42;
12 free_integer (p); -- p vaut 'null'
13 p.all := 1;
```

```
raised CONSTRAINT_ERROR : main.adb:13 access check failed
```

Toutefois, les désallocations demeurent dangereuses, notamment lorsque des alias sont utilisés, car dans ce cas la valeur `null` n'est pas propagée. Par exemple, le code suivant déclare deux accès de même type et les fait accéder vers le même objet :

```
type integer_access is access all integer;
p1, p2 : integer_access;
```

```
p1 := new integer;
p2 := p1;
p2.all := 42; -- legal
```

La désallocation du premier accès place la valeur `null` dans la variable mais pas dans son alias. Le second accès pourra ainsi être déréférencé bien qu'il soit invalide.

```
free_integer (p1); -- p1 vaut 'null', mais pas p2 !
put (integer'image (p2.all));
```

Les fonctions `new` et `ada.unchecked_deallocation` reposent sur la présence d'un mécanisme d'allocations dynamiques qu'il a fallu développer pour notre noyau. La sécurité des allocations repose donc sur la correction de l'allocateur sous-jacent.

Le code de cet allocateur, et du gestionnaire de mémoire dans son ensemble, est l'un élément les plus complexe du noyau. Le code Ada responsable de la gestion de la mémoire représente 2 500 lignes de codes, soit environ le quart du code total. Ce gestionnaire est composé de plusieurs paquetages qui gèrent l'allocation de pages physiques et de pages virtuelles, la gestion des défauts de page (*pages fault*) et l'allocation d'objets de taille arbitraire piochés dans des caches implémentés sous forme de *slabs*.

La gestion des allocations dynamiques pose plusieurs problèmes de conception et de sécurité et Ada ne protège que partiellement notre noyau contre des erreurs à ce niveau. Par ailleurs, nous avons vu que la prévention du déréférencement de pointeurs invalides ne fonctionne pas dans tous les cas.

Utilisation de contre-mesures *ad hoc* Nous avons vu que le typage permet de protéger les accès aux données entre l'espace d'adressage du noyau et l'espace d'adressage utilisateur. Toutefois, le typage ne permet pas de protéger contre l'exécution de code en zone de donnée ou bien contre des écritures accidentelles dans la zone de texte. Notre noyau utilise donc le support apporté par la pagination et certaines extensions matérielles[5] pour implémenter le W^X (*write or execute*).

5. En particulier les bits `NX` et `WP`.

4.7 Le coût en performance

Nous avons vu en section 4.3 que le compilateur Ada peut ajouter de nombreuses instructions en assembleur pour vérifier dynamiquement le respect du typage. Par conséquent, une pénalité est attendue au niveau des performances. Dans cette section, nous étudions les différences de performance entre du code en Ada et du code équivalent en C.

Deux études ont été faites sur le sujet avec des résultats différents. La première étude, qui porte sur le développement d'un logiciel embarqué [32] (1995) montre de meilleures performances pour le code en Ada, alors que la seconde étude, qui s'appuie sur l'outil de benchmarks *Dhrystone* [38] (2003), montre que le code en Ada est jusqu'à 3 fois plus lent que le même code en C.

Faute de résultats concordant, nous avons procédé à notre propre évaluation en nous appuyant sur deux tests.

Méthodologie Le premier test, réalisé entièrement en espace utilisateur sur une machine Linux, mesure le nombre de cycles d'horloge mis par une fonction qui réalise un tri bulle sur un tableau. Cette fonction a été implémentée en Ada et en C. Nous avons veillé à ce que chaque code utilise strictement le même algorithme. Les résultats ont été obtenus sur une moyenne de 10 000 mesures grâce à l'instruction assembleur `rdtsc`.

Le second test, effectué sur l'émulateur *Bochs*, permet de comparer le temps d'exécution d'un même appel système réalisé en C, d'une part, et en Ada, d'autre part. Les cycles ont été collectés grâce à l'instruction assembleur `rdtsc`. Bochs fournit un temps déterministe en considèrant qu'une instruction assembleur dure 1 cycle, avec pour conséquence que le nombre de cycles mesurés est constant pour un même binaire.

Pour valider l'utilisation de Bochs comme socle d'expérimentation, nous avons pris deux binaires et nous les avons fait s'exécuter en espace utilisateur sur une machine Linux puis dans Bochs (chargés par notre noyau). Le rapport des cycles d'horloge obtenus était dans les deux cas sensiblement le même.

Comparaison sur un tri bulle L'idée du premier test est de manipuler un type de données qui génère en Ada de nombreuses vérifications de typage à l'exécution. L'utilisation de tableaux est un choix approprié car en Ada, l'accès à un tableau donne lieu à un test pour vérifier l'absence de *buffer overflow*. Le choix du tri bulle a été guidé par le fait qu'il s'agit d'un algorithme inefficace, générant un grand nombre d'accès au tableau

à trier. Pour cette raison, mais aussi pour uniformiser la comparaison, le tableau de départ est inversement trié.

Le temps d'exécution a été mesuré en nombre de cycles d'horloge. Les résultats apparaissent dans le graphique en figure 2.

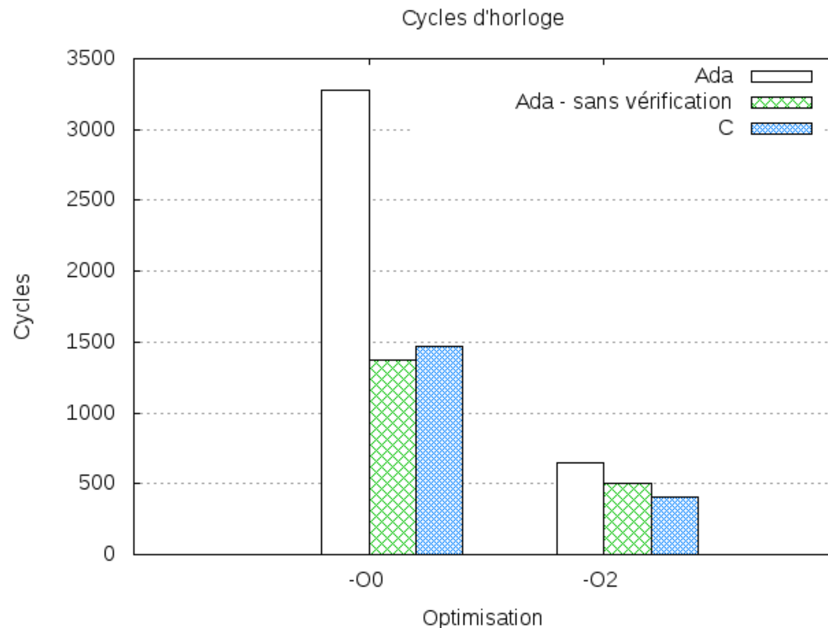

FIGURE 2. Tri bulle, comparaison code C / code Ada en cycles d'horloge

Ce graphique montre une différence nette de performance entre le code Ada, quand l'ensemble des vérifications sont activées, et le code C. En l'absence de toute optimisation (`-O0`) le code Ada utilise +122% de cycles d'horloge supplémentaires par rapport à son équivalent en C. Avec un niveau d'optimisation plus poussé (`-O2`), le code Ada est toujours plus lent avec +60% de cycles d'horloge supplémentaires.

On remarque que quand le code Ada est compilé avec une empreinte minimale à l'exécution, c'est-à-dire sans vérification de typage, sa performance se rapproche de celle du C (+24% de cycles), même si la différence reste marquée. Les raisons de cette différence de performance sont éclaircies en section 4.7, ou nous montrons que le compilateur gcc est plus optimisé pour le langage C que pour le langage Ada.

Ce premier test montre des performances relativement dégradées du code Ada par rapport au même code en C. Toutefois, ce test a été délibérément conçu comme étant un pire cas en défaveur de Ada. Cette contre-performance du code en Ada peut, de ce fait, être relativisée.

Comparaison sur un appel système Le second test compare l'exécution de l'appel système `syscall_print` en C et en Ada. Le choix de cet appel est intéressant car comme le test précédent, il manipule des chaînes de caractères et des tableaux. Initialement écrit en Ada, il a donc fallu réécrire l'ensemble de cet appel en C. Cela a d'ailleurs été l'occasion d'expérimenter l'interopérabilité entre C et Ada (car si les procédures ont été réécrites en C, la définition des structures de données demeurait en Ada).

Les mesures du nombre de cycles d'horloge effectués apparaissent dans le graphique en figure 3.

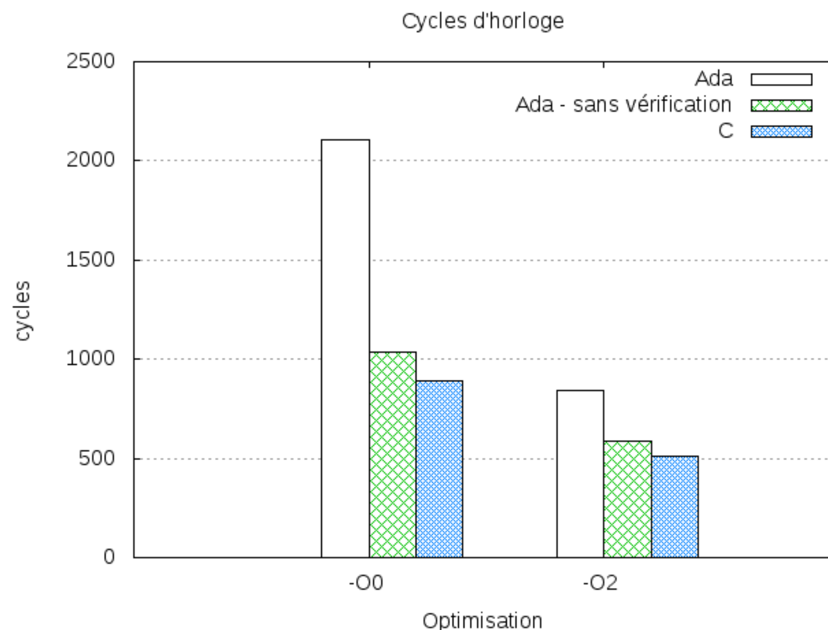

FIGURE 3. Appel système, comparaison code C / code Ada en cycles d'horloge

L'allure générale des courbes est similaire à ce qui a été observé avec le tri bulle (figure 2). Sans grande surprise, le code non optimisé (`-O0`) est plus lent que le code optimisé (`-O2`). Dans le premier cas, le code Ada avec l'ensemble des vérifications de typage activées est +137% plus lent que le code C équivalent. En optimisant la compilation (`-O2`), le code Ada est toujours plus lent, mais avec là encore +66% de cycles d'horloge supplémentaires. Avec la désactivation des vérifications de typages, les performances du code Ada se rapprochent de celles du code en C sans toutefois les égaler (+16% en `-O0` et +15% en `-O2`).

Comparaison de l'empreinte en mémoire Pour un noyau, un facteur de performance important est la taille de celui-ci, car elle impacte directement le taux de défauts de cache (*cache miss*) d'instructions et de données du processeur. Même sans pouvoir évaluer cet impact, une donnée intéressante concerne donc la différence de taille d'un binaire généré à partir de code Ada par rapport à son équivalent en C.

La figure 4 compare la taille en octets de la zone de texte des binaires dans le cadre de l'expérimentation sur le tri-bulle.

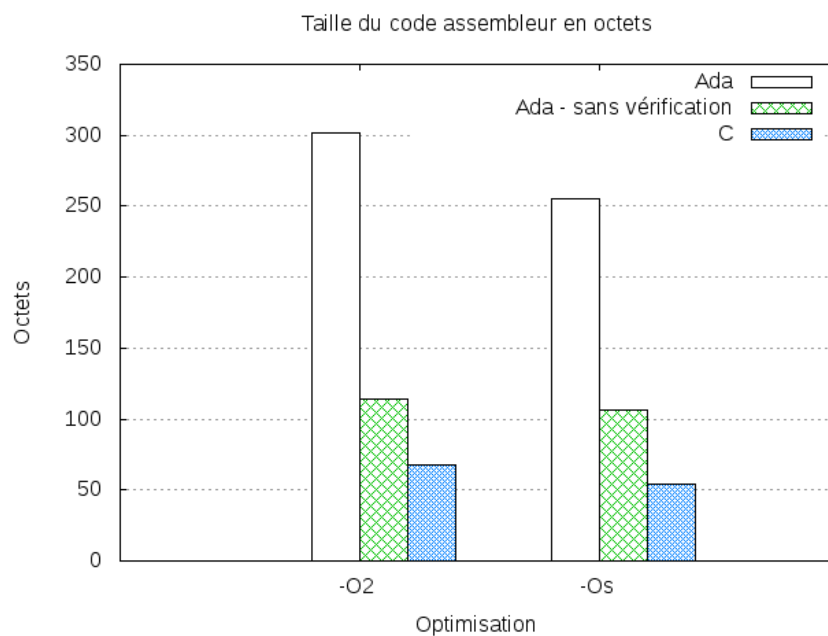

FIGURE 4. Tri bulle, comparaison code C / code Ada en taille

L'allure générale des courbes montre une différence très importante entre le code C et le code Ada. Lorsque les vérifications sont activées, le binaire généré à partir du code en Ada est environ 4 fois plus volumineux que son équivalent en C.

Avec la désactivation des vérifications de typages, le binaire généré à partir du code en Ada est de +68% (en optimisation `-02`) à +96% (en `-0s`) plus volumineux. Ce dernier résultat est le signe supplémentaire d'un manque d'optimisation de gcc pour le code en Ada.

Performance du code Ada Les résultats obtenus grâce à nos deux expérimentations sont concordantes. Dans les deux cas, nous nous sommes volontairement placé dans un cas défavorable à Ada en nous appuyant

sur des manipulations de tableaux. Sur du code optimisé (-O2), le code Ada avec l'ensemble des vérifications de typage activées subit une pénalité de +60% à +66% de cycles d'horloge supplémentaires. Avec le même niveau d'optimisation mais sans vérification dynamique, le code Ada subit une pénalité de +15% à +24%. Cette dernière peut être imputée à un manque d'optimisation de gcc qui produit alors un code presque deux fois plus volumineux.

5 Conclusion

Dans cette étude, nous avons démontré la faisabilité du développement d'un noyau en Ada, ce qui n'est pas surprenant car ce langage a été conçu dès l'origine pour le développement bas niveau. Si le noyau que nous avons développé est relativement rudimentaire, en partie du fait de l'effort conséquent à fournir pour supporter correctement les périphériques actuels, celui-ci nous a permis de démontrer la pertinence du langage Ada dans ce type de développement.

La mise en place de protections contre les vulnérabilités de type débordement ou pointeur invalide, grâce aux vérifications statiques et dynamiques, permettent de protéger le noyau tout en rendant visibles ces erreurs. Leur identification précise rend alors le débogage trivial.

L'un des points les plus importants concerne le coût de ces vérifications. Les résultats de tests portant sur des cas de figure défavorables au langage Ada montrent une pénalité maximum d'environ 60% de cycles d'horloge supplémentaires. Nous avons montré que la désactivation complète des vérifications à l'exécution permet d'obtenir des performances proches de celle du code C. La désactivation de ces vérifications peut s'avérer dangereuse si elle concerne autre chose que du code dont le fonctionnement est prouvé. Or notre étude nous a montré la simplicité de l'utilisation de Spark pour adapter certaines parties de code initialement en Ada.

D'autres études sur la sécurité des applications et des systèmes par l'utilisation de langages sécurisés existent. Dans chacune d'elles, les équipes ont créé de toute pièce de nouveaux langages ou dialectes (dérivés du C ou de l'assembleur) pour obtenir au final des garanties déjà fournies par le langage Ada.

Le mot de la fin revient à Tony Hoare. Lors de sa réception du prix Turing, il insista sur l'importance des vérifications dynamiques, qu'il implémenta pour le langage Algol60, afin de prévenir le dysfonctionnement des programmes critiques [16] :

It was logically impossible for any source language program to cause the computer to run wild, either at compile time or at run time. A consequence of this principle is that every occurrence of every subscript of every subscripted variable was on every occasion checked at run time against both the upper and the lower declared bounds of the array. Many years later we asked our customers whether they wished us to provide an option to switch off these checks in the interests of efficiency on production runs. Unanimously, they urged us not to - they already knew how frequently subscript errors occur on production runs where failure to detect them could be disastrous. I note with fear and horror that even in 1980, language designers and users have not learned this lesson.

Références

1. Ada 2012 Reference Manual, Language and standard libraries. `http://www.ada-auth.org/standards/ada12.htm`.
2. Ada 2005 Reference Manual, Language and standard libraries, 2006.
3. ADACore. SPARK Documentation. `http://docs.adacore.com/sparkdocs-docs/index.html`.
4. P. Argyroudis and Glynos D. Protecting the Core : Kernel Exploitation Mitigations. In *Black Hat Europe*, 2011.
5. John Barnes. *SPARK - The proven approach to high integrity software*. Altran Praxis, 2012.
6. Haogang Chen, Yandong Mao, Xi Wang, Dong Zhou, Nickolai Zeldovich, and M. Frans Kaashoek. Linux kernel vulnerabilities : state-of-the-art defenses and open problems. In *APSys '11*, 2011.
7. Aaron Ray Coleman. *Security kernel design for a microprocessor-based, multilevel, archival storage system*. PhD thesis, Naval Postgraduate School, 1979.
8. Common Vulnerabilities and Exposures website. CVE-2012-3364. Near Field Communication Controller Interface vulnerability in the Linux kernel.
9. Common Vulnerabilities and Exposures website. CVE-2012-4786. TrueType font parsing vulnerability in Windows kernels.
10. C. Cowan, C. Pu, D. Maier, H. Hintongif, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. StackGuard : Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proceedings of the 7th USENIX Security Symposium*, 1998.
11. CVE. Common Vulnerability and Exposure database. `http://cve.mitre.org/`.
12. Solar Designer. Linux kernel patch to remove stack exec permission, 1997.
13. Manuel Fahndrich and Robert DeLine. Adoption and focus : practical linear types for imperative programming. *SIGPLAN Not.*, 2002.
14. Michael Feldman. Who's using Ada ? `http://www.seas.gwu.edu/~mfeldman/ada-project-summary.html/`.

15. E. L. Glaser, J. F. Couleur, and G. A. Oliver. System design of a computer for time sharing applications. In *Proceedings of the November 30–December 1, 1965, fall joint computer conference, part I*, AFIPS '65 (Fall, part I), New York, NY, USA, 1965. ACM.
16. Charles Antony Richard Hoare. The emperor's old clothes. In *ACM Turing Award Lectures*, 1980.
17. Intel Corporation. Intel 64 and IA-32 architecture developer's manual.
18. Trevor Jim, J. Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. Cyclone : A Safe Dialect of C. In *Proceedings of the General Track of the annual conference on USENIX Annual Technical Conference*, ATEC '02, 2002.
19. Gerwin Klein. Correct OS kernel ? Proof ? Done ! *login*, 2009.
20. Jochen Liedtke. Improving IPC by kernel design. In *ACM Symposium on Operating System Principles*, 1993.
21. George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. CCured : type-safe retrofitting of legacy software. *ACM Trans. Program. Lang. Syst.*, 27(3), May 2005.
22. Yutaka Oiwa, Tatsurou Sekiguchi, Eijiro Sumii, and Akinori Yonezawa. Fail-safe ANSI-C compiler : an approach to making C programs secure. In *International Symposium on Software Security*, 2002.
23. Li Peng. Safe Systems Programming Languages, 2004.
24. Benjamin C. Pierce. *Types and programming languages*. MIT Press, 2002.
25. Dennis M. Ritchie. The development of the C language, 1993.
26. J. M. Rushby. Design and verification of secure systems. In *ACM Symposium on Operating System Principles*, 1981.
27. J. Rutkowska. On formally verified microkernels (and on attacking them). `http://theinvisiblethings.blogspot.fr/2010/05/on-formally-verified-microkernels-and.html`, 2010.
28. W. L. Schiller. Design of a Security Kernel for the PDP-11/45, December 1973.
29. Brad Spengler. Announcing UDEREF/amd64. `http://grsecurity.net/pipermail/grsecurity/2010-{A}pril/001024.html`.
30. Brad Spengler. Pax and Grsecurity. `http://grsecurity.net/`.
31. Swaroop Sridhar and Jonathan Shapiro. Bitc : a safe systems programming language, 2009.
32. David Syek. C vs Ada : arguing performance religion. *Ada Letters*, XV, 1995.
33. Andrew S. Tanenbaum, Jorrit N. Herder, and Herbert Bos. Can We Make Operating Systems Reliable and Secure ? *Computer*, 39(5), May 2006.
34. AMD64 Technology. AMD64 Architecture Programmer's Manual.
35. The Pax Team. Design & Implementation of PAGEEXEC, 2000.
36. Victor Van der Veen, Nitish Dutt Sharma, Lorenzo Cavallaro, and Herbert Bos. Memory errors : the past, the present, and the future. In *RAID'12*, 2012.
37. David A. Watt, Brian A. Wichman, and William Findlay. *Ada : language and methodology*. Prentice Hall International, 1987.
38. Marcus Weiskirchner. Comparison of the Execution Times of Ada, C and Java, 2003.