Compartimentation dynamique imbriquée pour objets contraints

Nicolas Dejon, Chrystel Gaber, Gilles Grimaud

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

Compartimentation dynamique imbriquée pour objets contraints

Nicolas Dejon[†‡], Chrystel Gaber[†] et Gilles Grimaud[‡]

[†]Orange Labs,
Châtillon, France
prénom.nom@orange.com

[‡]Univ. Lille, CNRS, Centrale Lille, UMR 9189
CRIStAL - Centre de Recherche en Informatique Signal et Automatique de Lille,
F-59000 Lille, France
prénom.nom@univ-lille.fr

Résumé
Ce papier présente un *framework* pour mettre en place une compartimentation imbriquée pour objets contraints. Tous les espaces mémoire compartimentés sont protégés par la *Memory Protection Unit* (MPU). Ce *framework* permet de prendre en compte le dynamisme des nouvelles applications sur objets contraints telle que la création temporaire d'un sous-espace mémoire protégé. Nous appliquons ce *framework* sur trois noyaux existants et montrons une protection des ressources équivalente. Nous proposons une implémentation de ce *framework* par l'utilisation de la MPU afin de créer et gérer ces sous-espaces selon un modèle de permission dynamique.

Mots-clés : compartimentation imbriquée, objets contraints, MPU

---

1. Introduction

La croissance de l'Internet des Objets (IdO, ou *IoT* en anglais) donne naissance à de nombreux cas d'usages novateurs dans plusieurs secteurs verticaux (réseau intelligent, agriculture intelligente, ville intelligente...).
Néanmoins, les objets contraints [5] sont aussi plus vulnérables aux attaques à cause de leurs ressources limitées, ce qui est d'autant plus critique pour les objets connectés (*IoT*) car exposés directement à des attaquants distants.
Les opportunités commerciales pressantes et les challenges sécuritaires mettent à mal le développement des objets connectés. Les nouveaux cas d'usage nécessitent une plus grande flexibilité que ce que propose les systèmes actuels, tout en garantissant une protection mémoire forte. Dans cet article, nous démontrons la faisabilité d'inclure de la compartimentation imbriquée protégée par le matériel au sein d'objets contraints. Les sous-espaces mémoire protégés peuvent eux-mêmes contrôler leur propre espace à la manière d'un noyau de séparation [14].

2. Protection mémoire pour les objets contraints avec MPU : scénario cible et état de l'art

2.1. Scénario cible

Un objet connecté en cours d'exécution souhaite isoler une zone mémoire au sein de son propre espace mémoire, soit en créant un sous-espace. Il peut s'agir d'une application tierce qui n'est

pas de confiance ou, au contraire, dispose d'une fonctionnalité critique qui doit être protégée. Nous souhaitons protéger le sous-espace d'interférences externes non désirées ou bloquer toutes ses tentatives d'accéder à de la mémoire qui ne lui est pas attribuée.
Le scénario considère un objet connecté contraint [5] disposant d'une *Memory Protection Unit*.

2.2. Modèle d'attaquant

Nous considérons un attaquant distant puissant pouvant installer des composants logiciels sur un objet connecté. L'adversaire se situe hors de l'espace noyau considéré de confiance et l'objectif est qu'il ne l'atteigne jamais. Les attaques physiques et par canaux cachés sont écartées. Néanmoins, l'attaquant ne peut pas réaliser des attaques par fautes à distance puisque nous avons besoin de l'intégrité des composants matériels et nous avons comme hypothèse qu'ils sont fonctionnellement corrects et opérationnels.

2.3. La *Memory Protection Unit* ou MPU

La MPU [1] est une unité facultative de certains processeurs ARM Cortex-M. Son rôle est de limiter les accès mémoire conformément à une configuration définie. Cette configuration comprend un jeu de blocs mémoire, appelés régions MPU, auxquels sont associés des permissions d'accès (lecture, écriture, exécution) et des attributs supplémentaires (de cache, de mise en tampon...). Un accès mémoire non autorisé résulte en une faute mémoire.
D'une façon générale, la MPU est pour les objets contraints ce que la *Memory Management Unit* (MMU) est pour les systèmes généralistes, puisque les deux partagent la fonctionnalité de protection mémoire. Néanmoins, la MPU est beaucoup plus contraignante puisque elle supporte au plus 16 régions MPU contre des millions de pages mémoire accessibles avec la MMU.

2.4. Etat de l'art

2.4.1. Systèmes statiques

Ces systèmes ne configurent qu'une unique fois la MPU, lors du démarrage du système, comme dans TrustLite [12]. La MPU produit donc une segmentation fixée et globale.

2.4.2. Systèmes dynamiques

Ces systèmes dédient la MPU à une protection locale, au grain d'un processus ou d'une fonction. Le principe est de reconfigurer la MPU à chaque activation d'un composant logiciel protégé, typiquement un changement de contexte. Le modèle de permission peut être soit immuable (fixé lors de la compilation ou du boot), ou bien variable (changé en cours d'exécution).
La majorité des systèmes ont un modèle de permissions immuable comme µVisor, ACES, MINION, EwoK, Choupi-OS, TockOS [2, 10, 11, 8, 4, 6]. Leurs différences tiennent dans le grain d'isolation proposé, les garanties et les politiques de sécurité mises en place, qui se transposent en autant de façons différentes de configurer la MPU.
D'autres systèmes ont un modèle de permission variable comme FreeRTOS-MPU [3] et Zephyr (MPU) [7]. Cependant, les régions MPU à disposition des applications sont limitées et la majorité des registres de configuration de la MPU sont réservés pour le système d'exploitation (SE). Par ailleurs, cette configuration applicative n'est pas restreinte et peut poser de sérieux problèmes de sécurité si elle n'est pas utilisée de façon sécurisée.
Peu de systèmes permettent d'étendre l'espace mémoire de ses composants au-delà de ce qu'il leur est initialement attribué. Les systèmes à modèle de permission variable peuvent être vus comme tels, mais sont extrêmement limités en possibilité (quelques régions au maximum).

2.4.3. Système flexibles

Les systèmes flexibles permettent une autre dimension de configuration de la MPU qui n'existe pas dans les sytèmes courants. Il n'y a aucun moyen pour ces systèmes de configurer la MPU

TABLE 1 – Comparaison de la compartimentation dans des systèmes basés MPU.

	Dynamisme			*Flexibilité*	
SE/noyau/ hyperviseur/outil	Reconfiguration MPU	Modèle de permission	Mémoire extensible	Nature de la compartimentation	Compartimentation imbriquée
`TrustLite`	✗	immuable	✗	processus	✗
`Choupi-OS`	✓	immuable	✗	processus	✗
`EwoK`	✓	immuable	✗	processus	✗
`µVisor`	✓	immuable	✗	*thread*	✗
`ACES`	✓	immuable	✗	générique	✗
`TockOS`	✓	immuable	(✓)	processus	✗
`Zephyr(MPU)`	✓	(variable)	(✓)	*thread*	✗
`FreeRTOS-MPU`	✓	(variable)	(✓)	tâche	✗
`framework` `proposé`	✓	variable	✓	générique	✓

autrement que le schéma d'attribution des ressources définis par le SE ou par instrumentation. C'est-à-dire que la configuration MPU est généralement prédéfinie avec au minimum certaines régions réservées pour des usages spécifiques, qui limitent par là même le nombre de régions MPU que peuvent utiliser les applications.
Les systèmes étudiés ont été classés selon ces critères dans la Table 1.

3. Compartimentation imbriquée avec la MPU

L'isolation proposée dans les systèmes actuels se comprend en horizontalité, avec des composants qui doivent être isolés entre eux. Nous proposons d'explorer l'isolation aussi en profondeur par de la compartimentation imbriquée. Nous présentons ici un *framework* pouvant servir à des systèmes basés sur MPU sans modification matérielle souhaitant de la flexibilité et sans compromis avec la sécurité.

3.1. Contrôle généralisé de la MPU

Nous proposons que n'importe quel composant, qu'importe son niveau d'abstraction (hyperviseur, SE, *thread*, processus...), ait la capacité de concevoir son espace mémoire interne comme il l'entend et en particulier de créer et de gérer une couche imbriquée (ses sous-espaces). Chaque composant à l'entière capacité de gérer son propre espace mémoire sans suivre des attributions de ressources fixées à l'avance et de pouvoir créer des sous-espaces ayant cette même capacité. L'espace mémoire d'un composant actif est gardé par la MPU. A chaque espace mémoire sa configuration MPU. Un composant peut prendre indirectement possession de la MPU pour isoler un composant interne (e.g. notre adversaire) et a la possibilité de modifier la configuration MPU. Toute tentative d'exfiltration de l'espace mémoire résultera en faute mémoire.

3.2. Configuration centralisée de la MPU par appel système

Les composants dans l'espace utilisateur, dont notre adversaire, ne doivent en aucune façon accéder à l'espace privilégié et prendre le contrôle du système. Cependant, la MPU doit être configurée par des opérations privilégiées depuis l'espace noyau. L'espace noyau se charge de réaliser ces opérations pour ces composants. Nous donnons à une entité dans l'espace noyau le rôle exclusif de configurer la MPU. Les composants peuvent ainsi modifier la configuration MPU via cette entité privilégiée.
Cette proposition va au-delà des régions mises à disposition de l'utilisateur dans FreeRTOS-MPU et Zephyr (MPU) puisqu'aucune des régions n'est réservée ou liée à des permissions particulières, pouvant être reconfigurées lors de l'exécution autant de fois que souhaité.
Nous définissons dans la section suivante une interface que cette entité doit implémenter.

3.3. Politique de sécurité personnalisée

Les composants doivent être limités dans leur capacité de configurer la MPU par l'intermédiaire de l'entité privilégiée. Autrement ce serait équivalent à leur donner des droits privilégiés

sur la mémoire et un composant malveillant pourrait s'octroyer plus de mémoire en prenant chez les autres composants par exemple. Cela signifie qu'il faut une restriction globale associée à la liberté de gérer son propre espace mémoire. La proposition inclut alors un cadre qui est mis en oeuvre par l'implémentation de l'interface. Par exemple, un composant pourrait créer un sous-espace et lui donner des instructions de traitement et des données. Cependant, il sera limité *par l'implémentation* de donner des droits supplémentaires aux blocs donnés que ceux initiaux, notamment pour respecter le principe $W \oplus X$ si cela fait partie de la politique de sécurité.

3.4. Interface de Programmation Applicative (IPA) adaptée pour la MPU

3.4.1. IPA dynamique

La plupart des systèmes étudiés reconfigurent la MPU à chaque changement de contexte. Cela permet de réutiliser des régions MPU pour d'autres processus plutôt que de les faire coexister tous en même temps. Nous sommes néanmoins intéressés par étendre la mémoire des processus lors de l'exécution et cela doit se faire sans considération du nombre de régions MPU sous-jacentes. Les blocs mémoire doivent donc être extensibles et modifiables.
L'IPA intègre six appels système permettant ce dynamisme :

- `add, remove` : ces appels système respectivement étendent et diminuent l'espace mémoire d'un composant logiciel. Ils peuvent être utilisés pour déplacer des blocs mémoire dans un sous-espace ou faire passer des messages.
- `prepare, collect` : ces appels système ont des fins de management. Ils doivent être appelés respectivement avant `add` et après `remove` afin de découpler les actes d'ajout et de récupération mémoire des opérations qui les rendent possibles (notamment la préparation de slots pour placer les nouveaux blocs ajoutés).
- `cut, merge` : ces appels système sont spécifiques aux systèmes sans blocs mémoire prédéfinis comme avec la MPU. Les opérations de découpe et de fusion permettent de redéfinir un bloc d'un espace mémoire, respectivement en coupant un bloc en deux ou en fusionnant deux blocs.

3.4.2. IPA flexible

L'IPA est augmentée de deux appels système pour créer et gérer un sous-espace :

- `create, delete` : ces appels système créent, respectivement détruisent, un sous-espace mémoire. La création consiste à déclarer un nouveau sous-espace. La destruction du sous-espace rend à l'espace mémoire qui gère le sous-espace toute la mémoire qu'il avait donné à ce sous-espace.

Ceci permet de casser les modèles de mémoire plate (*flat memory model*) et horizontaux en permettant d'établir une isolation en profondeur à l'initiative des applications.

4. Exemples de mise en place du *framework* pour Choupi-OS, TockOS et WooKey (EwoK)

Tous les SE utilisant la MPU le font pour protéger des ressources mais de façon bien différente comme vu dans la Section 2. Nous montrons ici que ces protections peuvent aussi être mises en place avec notre *framework* et qu'elles partagent des considérations communes.

Dans Choupi-OS, la ressource à protéger est le contexte de sécurité de l'applet active. Tout le reste de la mémoire dépend d'une configuration statique partagée entre contextes d'exécution. Dans notre *framework*, nous pouvons placer les contextes de sécurité dans des sous-espaces mémoire différents, tandis que les régions partagées seront copiées dans chacun des sous-espaces. Chaque activation de sous-espace aura ainsi exactement la même configuration (i.e. la même 'vue' mémoire) qu'actuellement. Le noyau Choupi a en revanche accès à toute la mémoire, qu'importe le contexte d'exécution. Il pourra donc faire partie de l'espace mémoire initial et les sous-espaces lui appartiendront, qu'il pourra créer lors de son lancement.

TockOS nécessite d'isoler des processus entiers et donc des blocs mémoire plus importants et plus nombreux. De la même manière qu'auparavant, nous isolons ces blocs dans des sous-espaces tandis que le SE est placé dans l'espace mémoire initial. Dans le cas présent il n'y a aucune mémoire partagée et donc chaque sous-espace dispose d'un jeu de blocs mémoire différents. TockOS permet aussi d'augmenter la mémoire allouée à un processus, ce qui est compatible avec notre IPA. Par ailleurs, TockOS charge et décharge des processus en mémoire en cours d'exécution, également possible avec notre *framework* par la création et la destruction de sous-espaces. Il est à noter dans TockOS la présence de pilotes de périphériques (appelés *capsules*) écrits par des tiers, actuellement placés dans l'espace noyau. TockOS et les pilotes sont tous écrits en Rust qui implique des accès surveillés et limités à leur espace propre, ce qui rend superflu l'usage de la MPU (la MPU est actuellement désactivée lors de leur exécution). De la même manière et pour les mêmes raisons, les pilotes peuvent rester dans l'espace noyau.

WooKey avec son noyau EwoK ne fait en revanche aucune concession sur les pilotes et les place dans l'espace utilisateur, isolés les uns des autres. Seules les parties de ces pilotes nécessitant une opération privilégiée (comme la configuration d'un périphérique) sont placées dans l'espace noyau. Ces parties ont été travaillées pour ne pas entrer en collision avec l'isolation mise en place par la MPU. Pareillement, nous plaçons ces pilotes dans des sous-espaces séparés de l'espace utilisateur et gardons les parties privilégiées de confiance dans le noyau. EwoK a la particularité supplémentaire de ne pas faire confiance totalement à son noyau et, bien qu'il ait les droits privilégiés, se retire tout accès aux code et données utilisateur lors de son exécution. Cela revient à placer le noyau (EwoK) également dans un sous-espace. L'espace mémoire initial ayant pour mission de placer le noyau dans ce sous-espace lors du boot, puis d'utiliser l'IPA définie pour gérer les sous-espaces des modules comme l'entend le noyau.

Nous pouvons en conclure que notre *framework* couvre les besoins actuels de protection par la MPU de ces noyaux. Nous observons une base commune pour les noyaux souhaitant protéger tout ou partie de leurs ressources applicatives. L'utilisation du *framework* permet en revanche une plus grande flexibilité qui peut se traduire pour EwoK en un nombre plus important de modules (ou pilotes pour WooKey), pour TockOS de protéger les *capsules* en les glissant hors du noyau et de pouvoir étendre la mémoire de chaque processus de façon beaucoup plus libre, ou encore de raffiner les protections dans Choupi voire même d'isoler le noyau des contextes s'exécutant comme fait pour EwoK. Par ailleurs, l'IPA permet une redéfinition des permissions d'accès pour chaque espace mémoire pouvant régler de façon encore plus fine leur politique de sécurité. De plus, nous extrayons des noyaux les composants responsables de la bonne configuration de la MPU qui est alors déléguée à notre entité privilégiée ce qui réduit leur besoin d'attention sur cette partie critique (sous couvert que notre entité privilégiée soit de confiance).

5. Implémentation de l'entité de gestion de compartimentation imbriquée par la MPU

Nous complétons la présentation de l'API par une proposition d'implémentation technique avec la MPU. Trois éléments sont essentiels pour guider la configuration de la MPU : la protection de l'espace mémoire, les structures qui permettent de définir un espace mémoire et leur isolation de l'espace utilisateur, et la virtualisation des régions MPU.

5.1. Protection de l'espace mémoire

L'espace mémoire d'un composant logiciel protégé est composé de plusieurs blocs mémoire. Chaque bloc est configuré comme une région MPU et doit donc être formé d'une adresse de début, d'une adresse de fin (ou une taille) et des permissions d'accès (lecture/écriture/exécution). Le changement de contexte se fait sans surcoût par rapport aux systèmes dynamiques.

5.2. Structures et protection des définitions d'espace mémoire

La définition d'un espace mémoire (i.e. les structures qui décrivent les blocs composant l'espace) doit être inaccessible pour l'espace utilisateur. Sinon, les composants non privilégiés pourraient directement modifier les structures et s'ajouter des privilèges ou de la mémoire.
La MPU peut protéger ces structures de plusieurs façons. Nous avons retenu l'option qui active la région d'arrière-plan (*background region* en anglais). La région d'arrière-plan correspond à la carte mémoire par défaut si la MPU est désactivée ou inexistante. En activant cette région, les composants privilégiés, telle que notre entité privilégiée, ont accès à tout l'espace mémoire hors ce qui est défini dans des régions MPU actives et applicable même aux composants privilégiés. Il suffit alors de retirer les régions de la MPU contenant les structures de définition pour que l'espace utilisateur en perde l'accès tout en restant accessible pour l'entité privilégiée. A noter que ces structures consomment directement la mémoire du composant actif ce qui limite son nombre total de sous-espaces et rend impossible un épuisement mémoire généralisé.

5.3. Virtualisation des régions MPU et algorithme de sélection des blocs mémoire

Un composant logiciel aura typiquement besoin de blocs mémoire pour ses code, données et périphériques. Une MPU de huit régions sera suffisante pour contenir tous ces blocs.
Néanmoins, les espaces mémoire peuvent être fragmentés. En particulier, ils peuvent contenir des blocs de sous-espaces qu'ils créeront plus tard ou des blocs de définition de sous-espaces. Ainsi, le nombre de blocs d'un espace mémoire est directement lié à la réduction de la fragmentation et aux états temporaires avant création de sous-espaces. Ce nombre va très probablement dépasser la capacité des huit régions MPU. Nous mettons alors en place une virtualisation de ces régions MPU (mais aucunement une virtualisation mémoire comme pour la MMU).
La MPU est tout d'abord initialisée avec certains blocs (actifs) de l'espace mémoire du composant actif. Puisqu'il peut y avoir plus de blocs mémoire que de régions MPU, certains blocs légitimes ne seront pas configurés dans la MPU au bon moment. S'ensuit une faute mémoire lors d'une tentative d'accès à ces blocs légitimes non configurés. Le noyau récupère cette faute et la transmet à l'algorithme de sélection de blocs qui va remplacer une des régions MPU par le bloc légitime qui a fauté. La MPU agit alors comme un cache. Plus le système sera fragmenté, plus il y aura de fautes mémoire. Dans une certaine mesure, cette virtualisation MPU est similaire à une opération de *copy-on-write* dans Linux ou à celle proposée pour les clés MPK [13].

6. Conclusion

De nouveaux cas d'usage révolutionnent les petits systèmes embarqués, nécessitant plus de complexité et de dynamisme que jamais. La création dynamique de sous-espaces mémoire protégés est un de ces cas d'usage mais n'est pas instanciable par les systèmes actuels sans modifications majeures. Nous proposons un *framework* basé sur l'usage de la MPU sans modifications matérielles permettant la mise en place d'une compartimentation imbriquée, adaptée aux cas d'usage émergents et assez flexible pour être compatible avec les systèmes actuels. Nous démontrons son utilisation pour protéger les mêmes ressources que ce que propose trois systèmes actuels ayant une forte appétence pour la sécurité. De plus, en l'utilisant, nous permettons ainsi à ces systèmes plus de flexibilité en terme de nombre de ressources protégées, de granularité de protection et de gestion dynamique des composants, sans jamais compromettre leur sécurité. Le rôle de configuration de la MPU est centralisé dans une unique entité privilégiée distincte. Nous proposons une virtualisation des régions MPU pour dépasser les contraintes matérielles fortes de la MPU. Au demeurant, les autres composants *de confiance* de l'espace noyau peuvent être buggés et violer involontairement la configuration MPU mise en place par notre entité privilégiée. Nous pourrions améliorer la sécurité en sortant ces composants de l'espace noyau et ne garder que notre entité privilégiée à la manière du protonoyau Pip [9].

Remerciements

Nous remercions chaleureusement l'ANSSI (Agence Nationale de la Sécurité des Systèmes d'Information) pour leur disponibilité et leur participation à l'étude préliminaire.
Toute opinion, conclusion ou recommandation exprimées dans cet article ne reflètent que celles des auteurs.

Bibliographie

1. ARM . – Website of : Armv8-m memory protection unit version 2.0. – https://static.docs.arm.com/100699/0200/armv8m_memory_protection_unit_100699_0200_en.pdf/, 2017. [Online ; accessed March 09, 2021].
2. ARMmbed . – Website of : The arm mbed uvisor. – https://github.com/ARMmbed/uvisor/, 2018. [Online ; accessed March 09, 2021].
3. FreeRTOS . – Website of : Freertos-mpu (freertos). – https://www.freertos.org/FreeRTOS-MPU-memory-protection-unit.html1, 2020. [Online ; accessed November 20, 2020].
4. Guillaume Bouffard and Léo Gaspard . – Website of : Choupi os (github). – https://github.com/gavz/choupi-os/, 2020. [Online ; accessed November 20, 2020].
5. IETF . – Website of : Terminology for constrained-node networks (ietf). – https://tools.ietf.org/rfc/rfc7228.txt, 2020. [Online ; accessed November 20, 2020].
6. Tock development team . – Website of : Tock. – https://www.tockos.org/, 2020. [Online ; accessed November 20, 2020].
7. Zephyr Project . – Website of : The zephyr project. – https://www.zephyrproject.org/, 2020. [Online ; accessed November 20, 2020].
8. Benadjila (R.), Michelizza (A.), Renard (M.), Thierry (P.) et Trebuchet (P.). – Wookey : Designing a trusted and efficient USB device. *ACM International Conference Proceeding Series*, 2019, pp. 673–686.
9. Bergougnoux (Q.), Iguchi-Cartigny (J.) et Grimaud (G.). – Pip , un proto-noyau fait pour renforcer la sécurité dans les objets connectés. *COMPAS*, 2017.
10. Clements (A. A.), Almakhdhub (N. S.), Bagchi (S.) et Payer (M.). – ACES : Automatic compartments for embedded systems. *Proceedings of the 27th USENIX Security Symposium*, 2018, pp. 65–82.
11. Kim (C. H.), Kim (T.), Choi (H.), Gu (Z.), Lee (B.), Zhang (X.) et Xu (D.). – Securing Real-Time Microcontroller Systems through Customized Memory View Switching. *NDSS*, 2018.
12. Koeberl (P.), Schulz (S.), Sadeghi (A. R.) et Varadharajan (V.). – TrustLite : A security architecture for tiny embedded devices. *Proceedings of the 9th European Conference on Computer Systems, EuroSys 2014*, 2014.
13. Park (S.), Lee (S.), Xu (W.), Moon (H.) et Kim (T.). – LibMPK : Software abstraction for intel memory protection keys (Intel MPK). *Proceedings of the 2019 USENIX Annual Technical Conference, USENIX ATC 2019*, 2019, pp. 241–254.
14. Rushby (J. M.). – Design and verification of secure systems. *Proceedings of the 8th ACM Symposium on Operating Systems Principles, SOSP 1981*, vol. 15, n5, 1981, pp. 12–21.