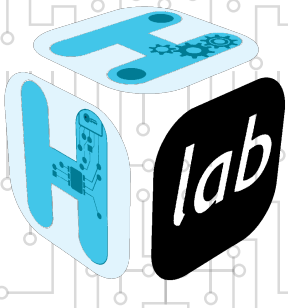
H2LAB-STIG-MPU-001
05-2026

Systèmes embarqués : guide de la protection mémoire physique



STIG H²Lab

Informations

Créée en 2020, H2Lab est une association loi 1901 dont l'objectif est de concevoir, développer et diffuser des solutions open-hardware et open-source dédiées à l'expérimentation autour des systèmes embarqués.
Ses principaux axes de travail sont :

- Conception de plateformes matérielles pour l'analyse hardware et software
- Développement d'alternatives ouvertes aux outils propriétaires, souvent opaques
- Publication de guides techniques et de recommandations pour promouvoir les bonnespratiques en matière de sécurité, de confidentialité et de durabilité dans l'écosystèmedes objets connectés et des systèmes embarqués

Soutien aux chercheurs, enseignants et étudiants via la mise à disposition d'outils, de contenus pédagogiques et de formations.
Partenariats académiques pour encourager la mutualisation des ressources et des savoirs.

Table des matières

1
Introduction et périmètre

1.1 Objectif du guide

Ce guide s'adresse prioritairement à deux profils :

- les ingénieurs cybersécurité embarquée qui doivent évaluer, justifier et auditer une stratégie de protection mémoire ;
- les développeurs bare-metal et experts firmware (avec ou sans RTOS) qui conçoivent et implémentent la configuration MPU au plus près du matériel.

Il est écrit pour répondre à un besoin opérationnel : passer d'un équipement sans protection mémoire à un système où la MPU est « utilisée comme contrôle de sécurité », avec une logique défendable en revue technique, en audit cyber et en validation produit. L'objectif n'est pas seulement de décrire les registres, mais de relier les mécanismes matériels à des menaces concrètes, à des choix d'architecture logicielle et à des critères de vérification.

Le guide couvre :

- les principes génériques de la MPU appliqués aux microcontrôleurs ;
- les variations importantes entre ARM Cortex-M, PowerPC 32 bits type MPCxx et RISC-V avec PMP ;
- la construction d'une politique de protection mémoire orientée cybersécurité ;
- les contre-mesures face aux attaques de corruption mémoire, d'exécution non autorisée et d'élévation de privilège ;
- la validation technique de la configuration MPU (tests, revues, critères d'acceptation).

Le guide ne couvre pas, ou seulement en interface :

- la sûreté de fonctionnement complète au sens normatif (ISO 26262, IEC 61508, etc.), hors points de contact avec la cybersécurité ;
- la sécurisation physique avancée (attaques invasives, fault injection matérielle avancée, side channels complexes) ;
- le détail exhaustif de chaque référence constructeur (manuels complets par variante), qui reste à traiter dans la documentation officielle de la cible matérielle.

En synthèse, ce document fournit un cadre technique pour concevoir, implémenter et vérifier une utilisation robuste de la MPU, en explicitant ce qu'elle permet de protéger, ce qu'elle ne permet pas de protéger, et comment l'intégrer dans une stratégie cyber embarquée cohérente.

2
Historique et évolution de la protection mémoire embarquée

2.1 Rappels historiques

Pendant longtemps, une grande partie des microcontrôleurs 8/16 bits a été conçue avec un modèle d'exécution monolithique : une seule image logicielle, un espace mémoire plat, et peu ou pas d'isolation matérielle entre fonctions critiques et non critiques. Ce modèle répondait aux contraintes de coût et de performance, mais il laissait un angle mort majeur : une corruption mémoire locale pouvait rapidement devenir une compromission globale du système.

L'introduction progressive de mécanismes de protection mémoire dans les cœurs embarqués a d'abord été motivée par la robustesse et la sûreté de fonctionnement, puis par la cybersécurité. Côté ARM, l'arrivée de la MPU sur les générations Cortex-M3/M4 puis Cortex-M7 a rendu possible une isolation régionale pragmatique sur des systèmes sans MMU complète [2, 4, 26, 27]. Côté automobile, les familles PowerPC e200 (par exemple NXP MPC57xx) ont structuré des approches de partitionnement mémoire et de contrôle d'accès adaptées aux contraintes temps réel dur, avec une forte proximité entre exigences safety et security [19, 20].

À partir de ARMv8-M, la combinaison MPU et séparation de domaine de sécurité (TrustZone-M) a marqué une étape importante : la protection mémoire n'est plus seulement une défense contre les fautes logicielles accidentelles, mais un mécanisme central de réduction de surface d'attaque entre mondes de confiance [5, 3]. Cette logique se retrouve dans des familles récentes comme STM32U5 ou NXP S32K3, où l'architecture vise explicitement l'intégration des exigences de cybersécurité produit [28, 21].

En parallèle, l'écosystème a convergé vers des concepts proches sur d'autres ISA embarquées. La protection physique mémoire (PMP) de RISC-V illustre cette convergence : granularité régionale, politiques d'accès par privilège, et ancrage dans une stratégie globale de durcissement [24, 25]. Cette trajectoire confirme un point clé pour l'ingénierie cyber : la protection mémoire embarquée est devenue un contrôle de base, au même niveau que le secure boot, la gestion des privilèges et la maîtrise des accès debug.

2.2 MPU vs MMU

La distinction MPU/MMU est déterminante pour cadrer ce que l'on peut exiger d'un microcontrôleur en cybersécurité.

Une MMU (Memory Management Unit) fournit classiquement de la traduction d'adresses virtuelle-vers-physique, de la pagination fine et un isolement fort entre processus, au prix d'une complexité matérielle et logicielle plus élevée (tables de pages, TLB, gestion mémoire virtuelle). Elle est adaptée aux systèmes riches (Linux, hyperviseurs), mais rarement pertinente pour la majorité des MCU temps réel à ressources contraintes.

Une MPU (Memory Protection Unit), à l'inverse, n'implémente pas de mémoire virtuelle. Elle applique des politiques d'accès sur des régions d'adresses physiques : lecture, écriture, exécution, niveau de privilège, et parfois attributs mémoire. Ce modèle est plus simple, déterministe et compatible avec les contraintes de latence des systèmes bare-metal/RTOS [2, 5, 6].

Sur le plan cybersécurité, cela implique :

- une excellente capacité à contenir les fautes et de nombreuses attaques de corruption mémoire *si* le partitionnement est rigoureux ;
- une dépendance forte à la qualité de configuration (taille des régions, priorités, politique par défaut) ;
- l'absence de certaines propriétés offertes nativement par une MMU (adressage virtuel, isolation processus fine par page).

En pratique, pour des plateformes telles que STM32F4/H7, SAM E70, S32K3 et MPC57xx, la MPU n'est pas une version « dégradée » de la MMU : c'est un mécanisme différent, optimisé pour l'embarqué critique, qui doit être conçu comme une barrière de confinement et de contrôle des privilèges au sein d'une architecture de défense en profondeur [26, 27, 17, 21, 19].

3

Rappels d'architecture : ce que protège réellement une MPU

3.1 Mécanismes fondamentaux

3.1.1 Régions mémoire

Le concept central de toute MPU est la *région* : un segment de l'espace d'adressage physique auquel sont associées des propriétés d'accès. Chaque région est définie par une adresse de base, une taille et un ensemble d'attributs. Le nombre de régions configurables simultanément est une contrainte matérielle fixe : 8 ou 16 pour la plupart des cœurs Cortex-M [2, 5], typiquement 16 à 24 pour les cœurs PowerPC e200 [19], et de 4 à 64 pour les implémentations RISC-V PMP [24].

Les contraintes d'alignement varient selon l'architecture : en ARMv7-M, chaque région doit être alignée sur un multiple de sa propre taille (contrainte puissance de deux), ce qui impose des compromis dans le découpage de l'espace mémoire [2, 4]. ARMv8-M lève cette contrainte en autorisant des régions alignées sur 32 octets, offrant une granularité sensiblement plus fine [5, 6]. Les implémentations PowerPC et RISC-V ont leurs propres règles de granularité minimale et d'alignement, à vérifier dans la documentation de chaque variante [19, 24].

Une région non configurée (ou dans une implémentation sans région de fond par défaut) déclenche une faute sur tout accès. Ce comportement *deny-by-default* est le mode le plus sûr : seule la mémoire explicitement autorisée est accessible. Son activation effective dépend de la politique de fond adoptée (voir section 3.2).

3.1.2 Permissions d'accès : lecture, écriture, exécution

À chaque région est associé un triplet de permissions :

- **Lecture (R)** : autorisation d'accès en lecture par le processeur ou un maître de bus.
- **Écriture (W)** : autorisation de modification du contenu de la région.
- **Exécution (X) ou *Execute Never* (XN)** : autorisation ou interdiction de récupérer des instructions depuis cette région. Le bit XN (ou son équivalent) est l'une des protections les

plus structurantes : il interdit qu'une zone de données soit utilisée comme zone de code, bloquant une classe entière d'exploitations.

En pratique, une politique robuste applique le principe de moindre privilège sur ces trois dimensions : les zones de code (Flash) sont marquées R+X mais non-W, les zones de données (RAM, pile) sont marquées R+W mais XN, et les périphériques mémoire mappés (MMIO) reçoivent uniquement les accès strictement nécessaires. Tout écart à ce principe constitue une surface d'attaque [2, 5].

3.1.3 Niveaux de privilège

La MPU opère en interaction directe avec les niveaux d'exécution du processeur. Sur Cortex-M, deux niveaux coexistent : *privilégié* (Privileged) et *non privilégié* (Unprivileged). Les attributs d'accès d'une région peuvent être différenciés selon ce niveau, permettant par exemple qu'une zone soit accessible en lecture-écriture par le code privilégié (noyau RTOS, pilotes) mais inaccessible ou en lecture seule pour le code utilisateur [2, 6].

Cette séparation est fondamentale pour la cybersécurité : elle traduit matériellement la frontière entre domaines de confiance. Sur ARMv8-M avec TrustZone-M, une troisième dimension s'ajoute — monde sécurisé / monde non sécurisé — permettant de cloisonner des actifs cryptographiques ou des secrets de provision au-delà de ce que la MPU seule peut offrir [5, 3].

Sur les cœurs PowerPC e200, la séparation superviseur/utilisateur (modes `PR` de `MSR`) joue un rôle analogue, avec des entrées de protection mémoire (IVOR, `SPROT` et attributs `UM`) définissant les droits par mode [19].

3.1.4 Recouvrements de régions et ordre de priorité

Lorsque plusieurs régions se chevauchent sur une même adresse, la MPU applique une règle de priorité déterministe. En ARMv7-M et ARMv8-M, la région de numéro le plus élevé l'emporte [2]. Cette règle permet de définir une région large avec des permissions par défaut, puis de superposer des régions plus étroites et plus restrictives pour des zones sensibles (par exemple, protéger en écriture une section de la RAM en marquant d'abord toute la RAM R+W puis en superposant une région XN+RO sur la table des vecteurs).

Ce mécanisme est puissant mais source d'erreurs : un recouvrement involontaire peut silencieusement annuler une restriction de sécurité. La vérification des recouvrements fait partie des revues de configuration obligatoires (voir chapitre 12). Il est cependant déconseillé d'utiliser ce type de méthodologie, ceci afin d'éviter tout effet de bord non maîtrisé. Il est préférable de découper les régions de manière non recouvrante, quitte à utiliser plusieurs régions pour couvrir une zone critique.

3.1.5 Exceptions de faute mémoire

Toute violation de la politique MPU déclenche une exception processeur. Sur Cortex-M, il s'agit du *MemManage Fault* (vecteur dédié dans la table NVIC), qui permet au gestionnaire d'exception d'analyser la cause via les registres `CFSR` (*Configurable Fault Status Register*) et `MMFAR` (*MemManage Fault Address Register*) [2, 26, 27].

Du point de vue cybersécurité, ce mécanisme est autant un outil de détection qu'une barrière. Un gestionnaire de faute correctement implémenté doit :

- enregistrer le contexte de la violation (adresse fautive, instruction incriminée, mode d'exécution) ;

- décider d'une réaction proportionnée : réinitialisation de la tâche, basculement en mode dégradé, arrêt sécurisé du système (*fail-safe*) ;
- ne jamais permettre la poursuite silencieuse de l'exécution après une violation, sauf si ce comportement est explicitement justifié et isolé.

Un gestionnaire de faute incomplet ou absent transforme une barrière de sécurité en simple mécanisme de détection sans réaction, ce qui est insuffisant dans un modèle de menace actif [5, 6].

Sur PowerPC e200, les violations de protection mémoire génèrent des exceptions *Data Storage* ou *Instruction Storage* (IVOR2/IVOR3), avec des registres de statut dédiés (`ESR`, `DEAR`) permettant une analyse similaire [19].

3.2 Limites intrinsèques

La MPU est un mécanisme de contrôle d'accès mémoire, pas un contrôle de cohérence logique. Il est essentiel de bien délimiter ce qu'elle protège réellement.

3.2.1 Ce que la MPU ne protège pas

Les attaques intra-région. La MPU ne distingue pas les accès légitimes des accès malveillants *au sein d'une même région*. Un débordement de tampon qui reste dans la même région de RAM ne sera pas détecté. La granularité de la protection est celle des régions : plus les régions sont larges, plus la protection est grossière.

La logique applicative. La MPU ne valide pas le comportement du code autorisé à s'exécuter. Un code légitime peut produire des effets indésirables (fuite d'information, usage incorrect d'une API) sans jamais déclencher de violation MPU. Elle ne remplace ni la validation des entrées, ni les vérifications logiques d'intégrité.

Les accès par DMA non contrôlé. La MPU processeur ne s'applique qu'aux accès initiés par le cœur. Les transferts DMA opèrent en tant que maîtres de bus indépendants : sans protection au niveau bus (XRDC, SMPU, MPU système selon les termes des constructeurs), un DMA mal configuré ou compromis peut atteindre des zones sensibles sans déclencher la MPU [19, 21]. Ce point est traité en détail au chapitre 11.

Les attaques physiques et les canaux auxiliaires. La MPU ne constitue aucune protection contre l'injection de fautes matérielles, les attaques par canal auxiliaire (analyse de consommation, EM) ou l'accès direct à la mémoire par des interfaces de debug non verrouillées.

Le code en dehors du modèle de protection. Si le chargeur de démarrage (*bootloader*) ou le code de démarrage (*startup*) s'exécute avant l'activation de la MPU, il n'est soumis à aucune restriction. La fenêtre temporelle avant activation est une surface d'attaque à traiter séparément (secure boot, verrouillage précoce).

3.2.2 Conditions nécessaires à l'efficacité réelle

La MPU n'est efficace que si plusieurs conditions sont simultanément remplies :

1. **Activation effective.** La MPU doit être explicitement activée (bit `ENABLE` dans `MPU_CTRL` sur Cortex-M). Elle est désactivée par défaut à la mise sous tension [2]. L'oubli de cette étape d'initialisation est l'une des erreurs les plus fréquentes.

2. **Politique de fond restrictive.** En l'absence de région correspondante, le comportement par défaut doit être l'interdiction totale (*background region* désactivée ou configurée comme *no access*). Une politique de fond permissive annule la valeur sécurisante de l'ensemble de la configuration.
3. **Cohérence de la configuration tout au long du cycle de vie.** La MPU peut être reconfigurée dynamiquement à l'exécution. Toute reconfiguration doit être réalisée en contexte privilégié et validée. Une reconfiguration non maîtrisée (par exemple via un vecteur d'attaque ayant obtenu l'exécution de code privilégié) peut démonter toute la politique de sécurité.
4. **Gestionnaire de faute opérationnel.** Sans gestionnaire de *MemManage Fault* implémenté et testé, les violations MPU déclenchent un comportement non spécifié ou un *HardFault* de repli, dont la réaction peut être incorrecte du point de vue sécurité.
5. **Synchronisation avec les autres mécanismes de protection.** La MPU doit être intégrée dans une stratégie plus large incluant le secure boot, le verrouillage des interfaces de debug et la gestion des privilèges RTOS. Isolée, elle constitue un contrôle nécessaire mais insuffisant.

En résumé, la MPU est une barrière structurelle efficace contre la propagation des corruptions mémoire et contre une large catégorie d'attaques de corruption de flux d'exécution, à condition que sa configuration soit rigoureuse, complète et maintenue sur l'ensemble du cycle de vie du produit [2, 5, 19, 24].

4

Modèle de menace pour systèmes embarqués MCU

4.1 Actifs, attaquants et vecteurs

4.1.1 Actifs à protéger

La construction d'un modèle de menace cohérent commence par l'identification des actifs dont la compromission entraîne un impact sécurité significatif. Pour un système embarqué MCU, on distingue cinq catégories d'actifs mémoire :

Code exécutable (firmware). L'image binaire stockée en Flash ou en ROM constitue l'actif fondamental : sa modification ou son remplacement partiel permet à un attaquant d'injecter un comportement arbitraire. La protection en écriture du code, combinée au secure boot, forme la première ligne de défense [2, 5].

Données de configuration et secrets. Clés cryptographiques, paramètres de calibration, identifiants de provision, certificats : ces données sont souvent stockées en Flash ou en RAM protégée. Leur lecture non autorisée permet l'usurpation d'identité ou le clonage de produit ; leur modification entraîne un comportement incorrect ou la désactivation des mécanismes de sécurité.

Structures de contrôle de l'exécution. La table des vecteurs d'interruption, la pile d'exécution (handler et thread), les structures internes du RTOS (blocs de contrôle de tâches, listes d'ordonnancement, mutex), les pointeurs de fonction : ces éléments orientent le flux d'exécution. Leur corruption est l'objectif principal des attaques de type *control-flow hijacking* [29, 11].

Mémoire des périphériques (MMIO). Les registres de configuration des périphériques, accessibles via des adresses fixes dans l'espace mémoire physique, permettent de contrôler le comportement matériel : activation du DMA, reconfiguration des horloges, accès aux interfaces de debug, modification des protections de Flash. Leur accessibilité non restreinte constitue un vecteur de pivot majeur dans les attaques sur systèmes industriels.

Région de mise à jour et de démarrage. Le bootloader et les mécanismes OTA sont des actifs de haute valeur : un attaquant capable d'en altérer le comportement obtient une persistance au redémarrage et un accès total au système. Les incidents Triton/TRISIS [12]

et Stuxnet [14] illustrent comment la compromission des couches basses d'un système de contrôle embarqué peut rester indétectée pendant des mois.

4.1.2 Profils d'attaquants

La littérature en sécurité des systèmes industriels et embarqués distingue généralement trois profils d'attaquants, caractérisés par leurs capacités d'accès et leurs moyens techniques [15, 13].

Attaquant distant (réseau ou bus). Il dispose d'une connexion aux interfaces de communication exposées : Ethernet industriel, CAN, LIN, BLE, Modbus/TCP, etc. Il n'a pas accès physique à l'équipement et son vecteur d'entrée passe obligatoirement par le code de traitement des messages reçus. Les travaux de Miller et Valasek sur la Jeep Cherokee (2015) [18] illustrent ce profil : l'accès initial par l'interface cellulaire Uconnect a permis d'atteindre le bus CAN interne puis de contrôler des unités de commande (ECU) sans aucun contact physique avec le véhicule. Cette taxonomie a été formalisée pour les systèmes embarqués automobiles par Checkoway *et al.* [9], dont l'analyse systématique des surfaces d'attaque reste une référence pour le threat modeling de systèmes embarqués connectés.

Attaquant avec accès physique limité. Il peut connecter des équipements sur des interfaces physiques exposées ou semi-exposées : connecteur JTAG/SWD, port UART de maintenance, interface CAN sur faisceau, port USB. Ses capacités incluent la lecture directe de la mémoire si les protections de debug ne sont pas activées, le débogage pas-à-pas et éventuellement l'injection de fautes légères. La recherche de Keen Security Lab sur les véhicules Tesla [16] illustre comment la combinaison d'un accès physique limité et d'interfaces insuffisamment verrouillées peut fournir un pivot vers des attaques distantes plus élaborées.

Attaquant co-localisé (code compromis). Il a déjà obtenu l'exécution de code non privilégié sur le MCU — par exploitation d'une vulnérabilité logicielle, injection de firmware ou compromission de la chaîne d'approvisionnement. Son objectif est l'escalade vers le contexte privilégié, la lecture d'actifs dans d'autres régions mémoire ou la modification de structures de contrôle. C'est le profil contre lequel la MPU offre la protection la plus directe, comme l'ont montré Clements *et al.* [11, 10] : sur des systèmes bare-metal représentatifs, l'absence de contrôle de privilège conduit systématiquement à l'exécution arbitraire complète après une corruption mémoire réussie.

4.1.3 Vecteurs d'attaque mémoire

Exploitation de vulnérabilités logicielles

Les vulnérabilités de corruption mémoire restent la famille d'attaque la plus documentée sur les systèmes embarqués. Szekeres *et al.* [29] ont formalisé cette classe dans leur systématisation : débordements de tampon sur pile (*stack buffer overflow*), dépassements de zone tas (*heap overflow*), déréférencements de pointeurs invalides et utilisations après libération (*use-after-free*). Sur les MCU bare-metal sans allocateur dynamique standard, les deux premières formes dominent largement.

La particularité des architectures MCU à espace d'adressage physique plat aggrave l'impact de ces vulnérabilités : une corruption réussie sur la pile peut directement écraser l'adresse de retour et rediriger l'exécution vers une zone arbitraire, sans qu'aucune barrière matérielle n'intervienne

en l'absence de MPU. Sur les implémentations sans isolation de pile, une seule écriture hors limites suffit à obtenir un contrôle total du flux d'exécution [29].

Interfaces de debug non verrouillées

Le maintien ouvert des interfaces JTAG/SWD constitue un vecteur de compromission complète, indépendant de toute vulnérabilité logicielle. Un attaquant disposant d'un accès physique à ces interfaces peut lire l'intégralité de la Flash et de la RAM, modifier l'exécution en temps réel et extraire les secrets de provision. De nombreux équipements industriels restent déployés avec les protections de lecture (*read-out protection*) au niveau minimal ou désactivées [13]. La recherche de Keen Security Lab [16] illustre comment ce vecteur peut être combiné avec une exploitation logicielle pour obtenir un accès persistant au système.

Même lorsque le *read-out protection* est activé, il ne faut pas le considérer comme un mécanisme absolu. Sur plusieurs familles STM32, des travaux publics ont montré des contournements pratiques du RDP de niveau intermédiaire (souvent *Level 1*) via injection de fautes (glitch tension/horloge), avec récupération partielle ou totale du contenu Flash selon la révision silicium et les contre-mesures activées [1, 22]. En revanche, les niveaux les plus stricts (ex. RDP2 sur certaines gammes) augmentent fortement le coût d'attaque mais au prix d'impacts opérationnels importants (débogage et maintenance). En pratique, le RDP doit donc être traité comme une brique de durcissement physique, à combiner avec secure boot, authentification de mise à jour et gestion stricte du cycle de vie des interfaces de debug [26, 27, 23].

Mécanismes de mise à jour

Les mécanismes OTA et les procédures de mise à jour en production constituent des vecteurs à fort impact. Une mise à jour non authentifiée ou dont la chaîne de vérification est contournable permet le remplacement du firmware. L'attaque Stuxnet [14] a démontré la faisabilité d'une modification ciblée du firmware d'un automate Siemens S7 à des fins de sabotage industriel discret, en exploitant une combinaison de vulnérabilités dans les outils d'ingénierie et les protocoles de mise à jour. L'incident Triton/TRISIS [12] a appliqué une logique similaire aux contrôleurs de sûreté Triconex de Schneider Electric, en injectant du code malveillant dans les modules de sécurité pour neutraliser les fonctions de protection de processus industriels critiques.

Pivots via DMA et périphériques

Dans les systèmes avec DMA ou plusieurs maîtres de bus, un périphérique compromis ou mal configuré peut accéder directement à la RAM sans passer par le cœur processeur, contournant intégralement la MPU processeur. Ce vecteur est traité en détail au chapitre 11.

4.1.4 Escalade de privilège : du vecteur d'entrée à la compromission mémoire

La chaîne d'exploitation typique sur un MCU sans protection mémoire suit un enchaînement prévisible :

1. **Accès initial** : exploitation d'une interface exposée (réseau, bus, debug) pour atteindre un code de parsing vulnérable.
2. **Corruption mémoire** : déclenchement d'un débordement de tampon ou d'une écriture hors limites, permettant d'atteindre une structure de contrôle (adresse de retour, pointeur de fonction, entrée de table de vecteurs).

3. **Redirection du flux d'exécution** : la structure corrompue oriente l'exécution vers un payload contrôlé par l'attaquant (code injecté en RAM ou chaîne ROP/JOP dans le code légitime) [29].
4. **Exécution arbitraire en contexte privilégié** : sur un MCU sans MPU, le code injecté hérite du niveau de privilège du contexte interrompu — souvent pleinement privilégié sur les systèmes bare-metal — donnant un accès total aux ressources système.

La MPU intervient comme barrière à deux niveaux de cette chaîne :

- entre les étapes 2 et 3, en restreignant les zones accessibles en écriture depuis le contexte courant (protection de la table des vecteurs, de la pile, des structures RTOS) ;
- entre les étapes 3 et 4, via le bit XN qui empêche l'exécution depuis les zones de données, bloquant l'injection directe de shellcode en RAM.

L'étude de Clements *et al.* [11] sur un corpus de binaires embarqués bare-metal confirme que la quasi-totalité des systèmes déployés sans MPU configurée offrent une surface d'exploitation triviale une fois le premier accès obtenu, et que l'activation d'un cloisonnement par privilège réduit significativement le périmètre atteignable à partir d'une tâche compromise.

4.2 Objectifs de sécurité

4.2.1 Formulation des objectifs de protection mémoire

À partir du modèle de menace précédent, on dérive un ensemble d'objectifs de sécurité directement adressables par la configuration de la MPU. Ces objectifs sont formulés de manière vérifiable : il doit être possible de les tester ou de les démontrer (voir chapitre 14).

OS-1 — Confinement des corruptions mémoire. Une corruption mémoire survenant dans le contexte d'un composant logiciel donné ne doit pas pouvoir modifier la mémoire appartenant à un autre composant ou au noyau/RTOS. *Mécanisme MPU concerné : partitionnement des régions RAM par domaine, permissions d'écriture restreintes par contexte d'exécution.*

OS-2 — Intégrité du flux d'exécution. Aucune zone de données (RAM, pile, heap, buffers de réception) ne doit pouvoir être utilisée comme source d'instructions exécutées. *Mécanisme MPU concerné : attribut XN systématique sur toutes les régions de données, autorisation d'exécution limitée aux zones de code en Flash.*

OS-3 — Protection des structures de contrôle critiques. La table des vecteurs d'interruption, les structures de contrôle du RTOS et les zones contenant des pointeurs de fonction ne doivent pas être modifiables depuis un contexte non privilégié ou depuis une tâche applicative isolée. *Mécanisme MPU concerné : permissions d'écriture réservées au contexte privilégié, recouvrements de régions restreignant l'accès en écriture sur les zones sensibles.*

OS-4 — Restriction des accès MMIO. L'accès aux registres de configuration des périphériques sensibles (contrôleur DMA, interfaces de debug, configuration de la Flash, registres de sécurité) doit être limité au code privilégié de bas niveau. *Mécanisme MPU concerné : régions MMIO en accès privilégié uniquement, interdiction d'accès depuis les tâches applicatives.*

OS-5 — Détection et réaction contrôlée. Toute tentative de violation des objectifs précédents doit déclencher une exception matérielle dont le traitement est déterministe, traçable et orienté vers un état sûr. *Mécanisme MPU concerné : gestionnaire MemManage Fault (ou équivalent) implémenté, testé et intégré à la stratégie de réponse.*

4.2.2 Correspondance avec le modèle de menace

Le tableau 4.1 établit la correspondance entre les profils d'attaquants identifiés, les vecteurs associés et les objectifs de sécurité adressés par la MPU.

Profil attaquant	**Vecteur d'attaque**	**Impact sans MPU**	**OS adressé**
Distant (réseau/-bus)	Parsing vulnérable (stack overflow)	Exécution arbitraire complète	OS-1, OS-2, OS-3
Distant (réseau/-bus)	Corruption de structures RTOS	Déni de service, élévation	OS-3
Accès physique	Interface debug non verrouillée	Lecture/écriture totale	Hors périmètre MPU
Accès physique	Mise à jour non authentifiée	Persistance malveillante	Hors périmètre MPU
Co-localisé	Escalade depuis tâche non privilégiée	Accès aux secrets, pivot noyau	OS-1, OS-3, OS-4
Système/DMA	Accès DMA non contrôlé	Contournement MPU processeur	OS-4 + protection bus

TABLE 4.1 – Correspondance entre profils d'attaquants, vecteurs et objectifs de sécurité adressés par la MPU

4.2.3 Périmètre résiduel non couvert par la MPU

Les objectifs OS-1 à OS-5 couvrent les menaces que la MPU peut adresser directement. Trois catégories de risques restent hors de ce périmètre et nécessitent des contrôles complémentaires (voir chapitre 15) :

- **Interfaces de debug et de mise à jour** : la sécurisation des interfaces JTAG/SWD, la désactivation ou l'authentification des accès debug, et la vérification cryptographique des images de mise à jour relèvent du secure boot et de la gestion du cycle de vie [5, 3]. Ces contrôles sont complémentaires à la MPU mais non substituables.
- **Attaques physiques avancées** : l'injection de fautes matérielles (glitching en tension ou en horloge), les attaques par canal auxiliaire et les attaques semi-invasives dépassent le modèle de protection de la MPU [13]. Elles relèvent de protections supplémentaires documentées dans les certifications de type CC EAL et dans les standards sectoriels [15].
- **Erreurs logiques dans le code autorisé** : un code correctement placé dans une région autorisée peut produire des effets indésirables (fuite d'information, usage incorrect d'une API, logique de contrôle défaillante) sans jamais déclencher de violation MPU. La validation fonctionnelle et les analyses statiques restent indispensables [29].

5
Cartographie des surfaces d'attaque mémoire

5.1 Zones critiques

La cartographie des surfaces d'attaque mémoire consiste à relier chaque zone d'adressage à trois questions opérationnelles : *qui peut y accéder*, *avec quels droits* et *quel impact en cas de compromission*. L'objectif n'est pas de dresser une liste exhaustive des adresses, mais de produire une vue exploitable pour la configuration MPU, les tests négatifs et la revue de sécurité [9, 11, 15].

Les zones à classifier en priorité sont :

- **Code exécutable** (bootloader, noyau, application) : risque d'altération en écriture et d'exécution hors périmètre prévu.
- **Données critiques** (clés, configuration de sécurité, état de contrôle) : risque de divulgation et de falsification.
- **Pile et zones dynamiques** : risque de corruption de flux de contrôle (retours, pointeurs de fonction, structures RTOS).
- **Table des vecteurs et handlers** : risque d'usurpation d'interruptions et de pivot vers du code privilégié.
- **MMIO et contrôleurs DMA** : risque de contournement de la logique applicative et d'accès indirect à la mémoire.
- **Interfaces de debug et de mise à jour** : risque de lecture/écriture globale du firmware hors chemin nominal.

5.1.1 Cas 1 : bare-metal séquentiel (un seul thread)

Dans une architecture bare-metal sans tâches applicatives concurrentes, les fonctions logicielles s'exécutent en séquence dans un même contexte de privilège. Cela réduit la complexité d'ordonnancement, mais concentre le risque : une corruption mémoire dans une fonction peut contaminer les fonctions suivantes, car elles partagent la même RAM globale et la même pile d'exécution.

La figure 5.1 met en évidence deux surfaces particulièrement sensibles en bare-metal :

- **les buffers RAM partagés** entre fonctions (acquisition, communication, diagnostic), qui deviennent des vecteurs de propagation transversale ;

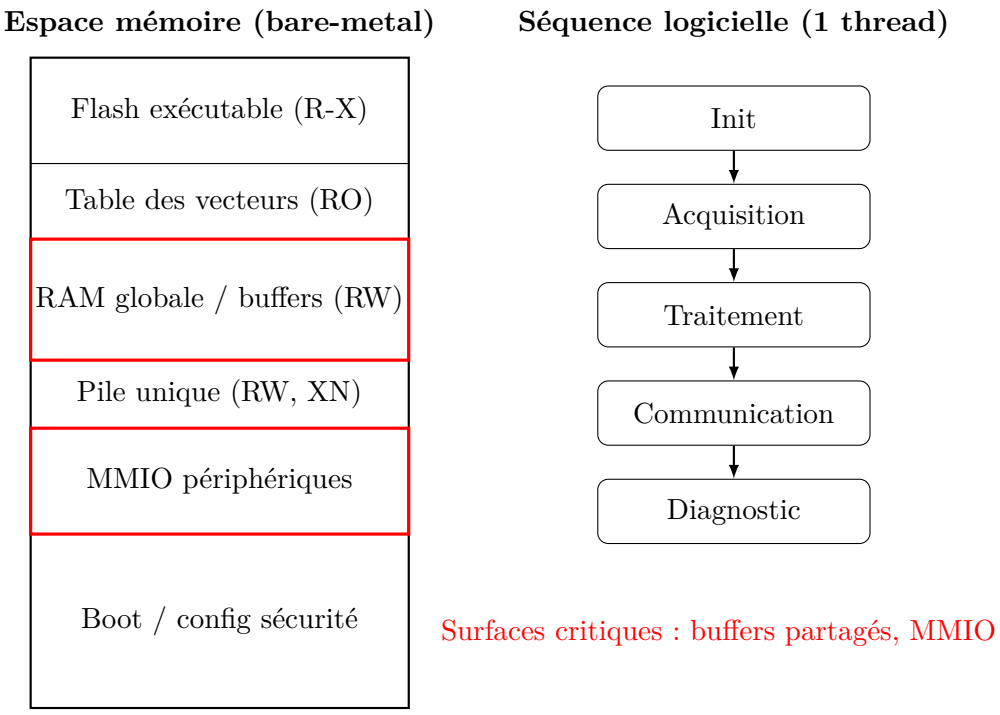


FIGURE 5.1 – Cartographie des surfaces d'attaque mémoire en mode bare-metal séquentiel

- **la zone MMIO**, où une écriture non contrôlée peut reconfigurer des périphériques critiques (clock, debug, DMA).

Pour ce profil, la cartographie doit identifier explicitement quelles fonctions manipulent ces zones, même en l'absence de multitache. Cela permet de découper la politique MPU en régions minimales (code, pile, données sensibles, MMIO) et d'éviter un *flat mapping* trop permissif.

5.1.2 Cas 2 : mode cloisonné avec applications (RTOS, multi-domaines)

En mode cloisonné, la surface d'attaque est plus large (plusieurs tâches, plus de transitions de contexte, davantage d'interfaces), mais elle devient plus gouvernable si chaque domaine applicatif dispose de son propre espace mémoire et de droits explicites. Le principe de base est de transformer une compromission locale en incident local, sans escalade systématique vers le noyau ni vers les autres applications [10, 5].

Les figures 5.2 et 5.3 illustrent les points de contrôle essentiels :

- **isolation App A / App B** en RAM (pas d'accès croisé direct) ;
- **passerelle d'appel privilégié** (SVC/syscall) comme unique voie vers les services noyau ;
- **reconfiguration MPU au changement de contexte** : profil noyau pendant le service, puis profil de la tâche planifiée ;
- **MMIO en liste blanche** avec séparation entre registres autorisés et interdits aux tâches non privilégiées ;
- **zone partagée minimale** pour les échanges inter-tâches, explicitement bornée et auditée.

Cette cartographie doit être maintenue au niveau exigence et au niveau implémentation (fichier de configuration MPU, lien script, revues de code), sans quoi le cloisonnement théorique dérive rapidement au fil des évolutions produit.

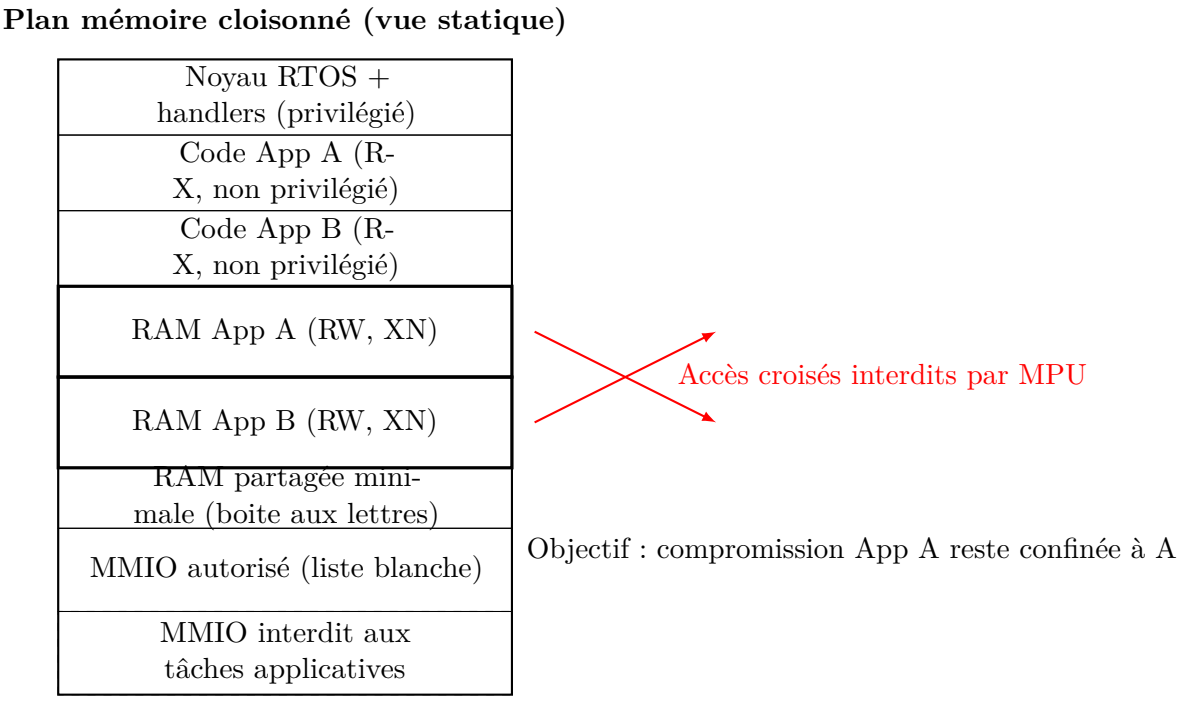


FIGURE 5.2 – Cartographie mémoire en mode cloisonné type RTOS

5.2 Priorisation du risque

Une fois les zones identifiées, la priorisation sert à décider où concentrer les efforts de durcissement et de validation. Une méthode simple et reproductible consiste à noter chaque zone selon trois axes :

- **Impact** : gravité métier et sécurité si la zone est compromise (de 1 à 5).
- **Exposition** : facilité d'atteinte depuis les vecteurs d'entrée réels (réseau, bus, debug, mise à jour) (de 1 à 5).
- **Exploitabilité** : effort technique pour transformer l'accès en compromission utile (de 1 à 5).

On peut utiliser un score initial $S = I \times E \times X$ pour établir un ordre de traitement, puis ajuster ce score par des facteurs contextuels (maturité des contre-mesures, capacité de détection, exposition terrain). L'objectif n'est pas la précision mathématique absolue, mais la cohérence décisionnelle entre équipes.

Zone mémoire	Impact	Exposition	Exploit.	Score
Table des vecteurs / handlers	5	3	4	60
RAM partagée inter-domaines	4	4	4	64
MMIO (DMA, debug, flash ctrl)	5	3	5	75
Code applicatif non critique	3	3	3	27
Zone de logs non sensible	1	4	2	8

TABLE 5.1 – Exemple de priorisation des surfaces d'attaque mémoire

Dans la pratique, une zone prioritaire doit déclencher trois actions minimales :

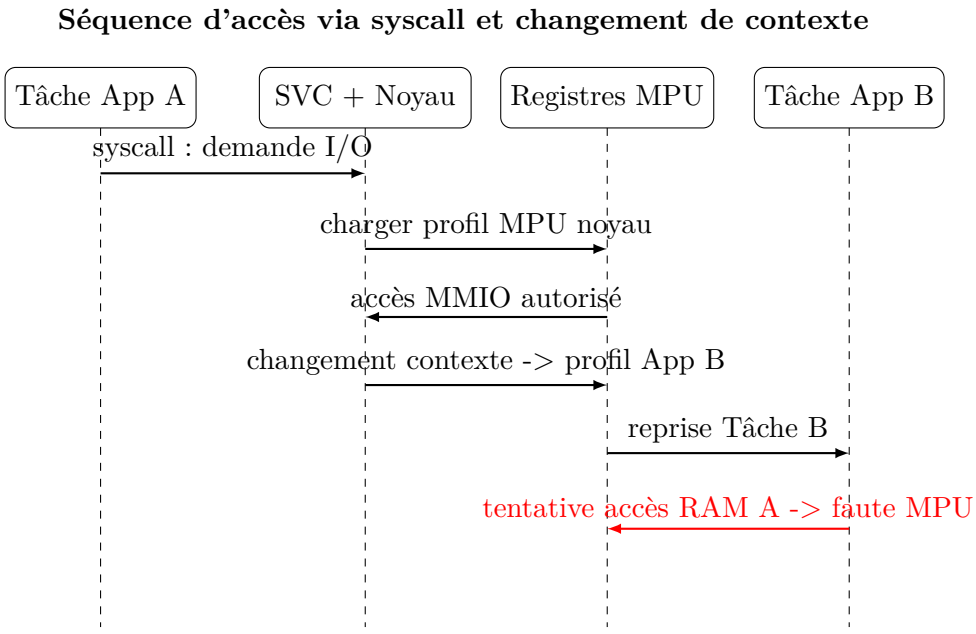


FIGURE 5.3 – Séquence simplifiée des syscalls et de la reconfiguration MPU au changement de contexte

1. **mesure de confinement** dans la configuration MPU (réduction de droits, séparation de régions, XN) ;
2. **test négatif dédié** (accès interdit en lecture/écriture/exécution) ;
3. **trace de justification** dans la documentation de sécurité (exigence, preuve de test, décision d'acceptation de risque résiduel).

Cette logique de priorisation aligne la cartographie avec les chapitres suivants : configuration MPU par architecture, stratégie d'isolation, contre-mesures, puis validation et non-régression sécurité.

6
MPU sur ARM Cortex-M : fonctionnement et variantes

6.1 Logique générique Cortex-M

Sur Cortex-M, la protection mémoire repose sur la *Protected Memory System Architecture* (PMSA), c'est-à-dire un modèle par régions d'adresses physiques, sans traduction virtuelle, appliqué au moment de chaque accès instruction ou donnée [2, 5]. La MPU est donc un filtre matériel entre le pipeline CPU et le bus mémoire :

1. le cœur calcule une adresse cible (fetch, load/store) ;
2. la logique MPU détermine la région correspondante (ou l'absence de région) ;
3. les droits de la région (R/W/X, privilège, attributs mémoire) sont comparés au contexte courant ;
4. en cas de violation, une exception *MemManage Fault* est générée.

Ce modèle est déterministe et adapté aux contraintes temps réel des MCU. La robustesse de la protection dépend directement du partitionnement en régions, de la politique par défaut et de la qualité des handlers de faute [4, 6].

Dans l'écosystème logiciel, CMSIS-Core expose une vue unifiée des registres MPU via la structure `MPU_Type` et des fonctions utilitaires (ex. `ARM_MPU_SetRegion`, `ARM_MPU_Enable`) [7, 8]. Les cartes de registres suivantes reprennent les registres typiques utilisés en pratique.

6.1.1 Exemple d'implémentation assembleur ARMv7-M (PMSAv7)

L'exemple suivant illustre une initialisation minimale en ARMv7-M : région Flash exécutable en lecture seule, région SRAM en lecture/écriture non exécutable, puis activation MPU avec barrières de synchronisation.

```
.syntax unified
.thumb

/* Registres MPU ARMv7-M */
.equ MPU_TYPE, 0xE000ED90
```

PMSA ARMv7-M : registres MPU typiques (CMSIS)

Offset	Nom	Rôle principal
0x00	TYPE	Capacités MPU (nombre de régions).
0x04	CTRL	Activation + politique défaut privilégiée.
0x08	RNR	Sélection de la région N à programmer.
0x0C	RBAR	Adresse de base de région (alignée).
0x10	RASR	Taille + AP + XN + attributs + sous-régions.
0x14	RBAR_A1	Alias RBAR pour chargement rapide.
0x18	RBAR_A2	Alias RBAR pour chargement rapide.
0x1C	RASR_A1	Alias RASR pour chargement rapide.
0x20	RASR_A2	Alias RASR pour chargement rapide.

Vue alignée sur `MPU_Type` CMSIS-Core (Cortex-M3/M4/M7).

FIGURE 6.1 – Registres MPU typiques en ARMv7-M selon la modélisation CMSIS-Core

```
.equ MPU_CTRL, 0xE000ED94
.equ MPU_RNR,  0xE000ED98
.equ MPU_RBAR, 0xE000ED9C
.equ MPU_RASR, 0xE000EDA0

/* Exemples d'attributs RASR */
.equ RASR_ENABLE,     (1 << 0)
.equ RASR_SIZE_1MB,   (19 << 1)       /* log2(1MB)-1 */
.equ RASR_SIZE_128KB,(16 << 1)       /* log2(128KB)-1 */
.equ RASR_AP_RO_ALL, (6 << 24)
.equ RASR_AP_RW_ALL, (3 << 24)
.equ RASR_XN,         (1 << 28)

mpu_init_v7m:
  /* 1) Desactiver MPU */
  ldr r0, =MPU_CTRL
  movs r1, #0
  str  r1, [r0]
  dsb
  isb

  /* 2) Region 0: Flash @0x08000000, RO, executable */
  ldr r0, =MPU_RNR
  movs r1, #0
  str  r1, [r0]

```

PMSA ARMv8-M : registres MPU typiques (CMSIS)

Offset	Nom	Rôle principal
0x00	TYPE	Capacités MPU (régions disponibles).
0x04	CTRL	Activation et politique par défaut.
0x08	RNR	Sélection de la région N.
0x0C	RBAR	Base + AP/XN/partage (PMSAv8).
0x10	RLAR	Limite + AttrIndx + enable.
0x14	RBAR_A1	Alias RBAR pour chargement multi-régions.
0x18	RLAR_A1	Alias RLAR (limite/attribut/index).
0x1C	RBAR_A2	Alias RBAR pour chargement multi-régions.
0x20	RLAR_A2	Alias RLAR (limite/attribut/index).
0x24	RBAR_A3	Alias RBAR pour chargement multi-régions.
0x28	RLAR_A3	Alias RLAR (limite/attribut/index).
0x30	MAIR0	Attributs mémoire index 0..3.
0x34	MAIR1	Attributs mémoire index 4..7.

Vue alignée sur `MPU_Type` CMSIS-Core (Cortex-M23/M33/M55).

FIGURE 6.2 – Registres MPU typiques en ARMv8-M (PMSAv8) selon la modélisation CMSIS-Core

```
32    ldr r0, =MPU_RBAR
33    ldr r1, =0x08000000
34    str  r1, [r0]
35
36    ldr r0, =MPU_RASR
37    ldr r1, =(RASR_ENABLE | RASR_SIZE_1MB | RASR_AP_RO_ALL)
38    str  r1, [r0]
39
40    /* 3) Region 1: SRAM @0x20000000, RW, XN */
41    ldr r0, =MPU_RNR
42    movs r1, #1
43    str  r1, [r0]
44
45    ldr r0, =MPU_RBAR
46    ldr r1, =0x20000000
47    str  r1, [r0]
48
49    ldr r0, =MPU_RASR
50    ldr r1, =(RASR_ENABLE | RASR_SIZE_128KB | RASR_AP_RW_ALL | RASR_XN)
51    str  r1, [r0]
52
53    /* 4) Activer MPU + default map privilegiee */
54    ldr r0, =MPU_CTRL
55    movs r1, #5                  /* ENABLE=1, PRIVDEFENA=1 */
56    str  r1, [r0]
```

```
  dsb
  isb
  bx    lr
```

Cet exemple est volontairement minimal : en production, il faut ajouter le durcissement des régions MMIO, la protection de la table des vecteurs, la stratégie handler/thread et des tests négatifs systématiques [2, 26, 27].

6.1.2 Exemple d'implémentation assembleur ARMv8-M (PMSAv8)

L'exemple ci-dessous montre le principe typique : programmation de `MAIR0`, création d'une région Flash (index attribut 0) et d'une région SRAM XN (index attribut 1), puis activation MPU.

```
.syntax unified
.thumb

/* Registres MPU ARMv8-M (PMSAv8) */
.equ MPU_CTRL,  0xE000ED94
.equ MPU_RNR,   0xE000ED98
.equ MPU_RBAR,  0xE000ED9C
.equ MPU_RLAR,  0xE000EDA0
.equ MPU_MAIR0, 0xE000EDC0

/* RBAR bits simplifies */
.equ RBAR_AP_RO_ALL, (2 << 1)
.equ RBAR_AP_RW_ALL, (1 << 1)
.equ RBAR_XN,        (1 << 0)

/* RLAR bits simplifies */
.equ RLAR_EN,        (1 << 0)
.equ RLAR_ATTR0,     (0 << 1)
.equ RLAR_ATTR1,     (1 << 1)

mpu_init_v8m:
  /* 1) Desactiver MPU */
  ldr r0, =MPU_CTRL
  movs r1, #0
  str  r1, [r0]
  dsb
  isb

  /* 2) MAIR0: Attr0=0xFF (Normal WBWA), Attr1=0x44 (Normal non-cache
   ) */
  ldr r0, =MPU_MAIR0
  ldr r1, =0x000044FF
  str  r1, [r0]

  /* 3) Region 0: Flash 0x08000000..0x080FFFFF, RO, executable, Attr0
    */
```

```
35    ldr r0, =MPU_RNR
36    movs r1, #0
37    str  r1, [r0]
38
39    ldr r0, =MPU_RBAR
40    ldr r1, =(0x08000000 | RBAR_AP_RO_ALL)
41    str  r1, [r0]
42
43    ldr r0, =MPU_RLAR
44    ldr r1, =(0x080FFFFF | RLAR_ATTR0 | RLAR_EN)
45    str  r1, [r0]
46
47    /* 4) Region 1: SRAM 0x20000000..0x2001FFFF, RW, XN, Attr1 */
48    ldr r0, =MPU_RNR
49    movs r1, #1
50    str  r1, [r0]
51
52    ldr r0, =MPU_RBAR
53    ldr r1, =(0x20000000 | RBAR_AP_RW_ALL | RBAR_XN)
54    str  r1, [r0]
55
56    ldr r0, =MPU_RLAR
57    ldr r1, =(0x2001FFFF | RLAR_ATTR1 | RLAR_EN)
58    str  r1, [r0]
59
60    /* 5) Activer MPU */
61    ldr r0, =MPU_CTRL
62    movs r1, #5                   /* ENABLE=1, PRIVDEFENA=1 */
63    str  r1, [r0]
64    dsb
65    isb
66    bx   lr
```

Le point clé de migration v7 vers v8 est de ne pas translater mécaniquement les encodages `RASR`. Il faut reconstruire explicitement les attributs mémoire (MAIR), la borne haute (RLAR) et la stratégie d'appel privilégié/syscalls pour conserver des propriétés de sécurité équivalentes [5, 8].

6.2 Variations selon architecture

Les différences principales entre ARMv7-M (PMSAv7) et ARMv8-M (PMSAv8) impactent directement la portabilité des politiques MPU.

En ARMv8-M, la présence de TrustZone-M ajoute une dimension de partitionnement supplémentaire : selon le cœur et l'intégration SoC, la configuration peut distinguer domaines sécurisés et non sécurisés, avec des règles spécifiques d'accès aux ressources partagées [5, 3, 6, 23].

Aspect	**ARMv7-M**	**ARMv8-M**
Définition de région	Base + taille (`RBAR`/`RASR`)	Base + limite (`RBAR`/`RLAR`)
Granularité	Forte contrainte d'alignement puissance de deux	Modèle plus flexible (limite explicite)
Attributs mémoire	Encodage dans `RASR` (TEX/C/B/S)	Index d'attributs via `MAIR0`/`MAIR1`
Sous-régions	Disponibles (désactivation par sous-régions)	Non utilisées comme en v7, logique de limite privilégiée
Sécurité TrustZone-M	Non	Oui (Secure/Non-secure selon implémentation)
CMSIS API	`ARM_MPU_*` v7	`ARM_MPU_*` v8 (RBAR/RLAR/MAIR)

Table 6.1 – Différences structurantes entre PMSAv7 et PMSAv8 pour la configuration MPU

7

MPU sur PowerPC MPC57xx : fonctionnement et variantes

7.1 MPU coeur e200 : principe et registres

Sur MPC57xx, la première couche de protection mémoire est la MPU *instanciée dans chaque coeur e200*. Concrètement, chaque coeur possède ses propres entrées de protection (programmées via les registres MAS/TLB) et applique les droits d'accès sur ses fetch instructions et ses load/store [19, 20].

Ce point est structurant : sur un SoC multicore, la politique MPU coeur doit être configurée pour **chaque coeur** au démarrage. Une configuration valide sur le coeur 0 ne protège pas automatiquement le coeur 1.

Le principe opérationnel est le suivant :

1. le coeur résout une entrée MPU/TLB correspondante à l'adresse accédée ;
2. l'entrée fournit taille de zone, attributs mémoire et permissions (R/W/X, privilège) ;
3. en cas de violation, le coeur déclenche une exception *Data Storage* ou *Instruction Storage*.

Cette MPU coeur couvre les accès initiés par le coeur e200. Les autres maîtres de bus (DMA, périphériques bus masters) nécessitent une protection complémentaire côté SMPU système.

7.1.1 Exemple d'implémentation assembleur de la MPU coeur

L'exemple ci-dessous illustre une programmation minimale d'une entrée de protection statique (TLB1) pour une zone Flash en lecture+exécution. Les encodages exacts de bits (taille, attributs WIMGE, permissions) doivent être adaptés au coeur e200 ciblé et à la politique produit [19].

```
/* Exemple simplifie e200 core MPU/TLB - syntaxe GNU as */

.equ MAS0_TLBSEL1_ESEL0,  0x10000000
.equ MAS1_VALID_1MB,      0x80000500   /* exemple: V=1, TSIZE=1MB */
```

Registre	Nom	Rôle principal
SPR 624	MAS0	Sélection de l'entrée TLB/MPU à programmer (bank, index, commande).
SPR 625	MAS1	Validité d'entrée, taille de zone, identifiant de contexte.
SPR 626	MAS2	Adresse virtuelle/effective de base et attributs mémoire (WIMGE).
SPR 627	MAS3	Adresse physique et droits d'accès (R/W/X superviseur et utilisateur).
SPR 944	MAS6	Sélection complémentaire de contexte/protection ID selon variante.
SPR 945	MAS7	Bits d'adresse haute pour variantes avec espace étendu.
SPR 1015	MMUCSR0	Contrôle MMU/MPU coeur, invalidation globale des entrées.
SPR 48	PID0	Identifiant de processus/contexte utilisé par le matching d'entrées.
SPR 688	TLB1CFG	Capacité TLB1 (nombre d'entrées statiques disponibles).

Noms et comportements exacts selon coeur e200 (z2/z4/z7) et variante MPC57xx.

FIGURE 7.1 – Registres coeur e200 utilisés pour programmer la MPU (MPC57xx)

```
.equ MAS2_EPN_FLASH,      0x00400000   /* + attributs WIMGE si
   necessaire */
.equ MAS3_RPN_FLASH_RX,   0x0040003D   /* exemple: SR/SX autorises,
   SW interdit */

mpu_core_init_mpc57xx:
  /* 1) Selection entree TLB1[0] */
  lis   r3, MAS0_TLBSEL1_ESEL0@h
  ori   r3, r3, MAS0_TLBSEL1_ESEL0@l
  mtspr MAS0, r3

  /* 2) Definir taille/validite et base effective */
  lis   r3, MAS1_VALID_1MB@h
  ori   r3, r3, MAS1_VALID_1MB@l
  mtspr MAS1, r3

  lis   r3, MAS2_EPN_FLASH@h
  ori   r3, r3, MAS2_EPN_FLASH@l
  mtspr MAS2, r3

  /* 3) Definir adresse physique + droits */
  lis   r3, MAS3_RPN_FLASH_RX@h
  ori   r3, r3, MAS3_RPN_FLASH_RX@l
  mtspr MAS3, r3

  /* 4) Ecrire l'entree dans le TLB */
  tlbwe
  msync
  isync
  blr
```

7.2 SMPU système : logique générique MPC57xx

Sur les familles NXP MPC57xx, la protection mémoire repose également sur un mécanisme de type SMPU (System Memory Protection Unit) associé à une logique multi-maîtres de bus. Le modèle reste régional (base/limite + droits), mais l'analyse sécurité doit tenir compte à la fois du coeur e200 et des autres maîtres (DMA, accélérateurs, périphériques bus masters) [19, 20].

La séquence type de vérification d'accès est la suivante :

1. un maître émet un accès instruction ou data ;
2. le SMPU sélectionne la région correspondante ;
3. les droits associés au domaine/maître sont comparés ;
4. en cas de refus, une exception (ou un événement de violation) est levée avec capture d'adresse et de statut.

Du point de vue cyber, l'intérêt majeur par rapport à une MPU purement CPU-centrique est la capacité à gouverner plusieurs initiateurs d'accès. Cela réduit le risque de contournement par voies indirectes, notamment via DMA, à condition de configurer explicitement les profils de chaque maître [19, 21].

Offset	Nom	Rôle principal
0x000	CESR	Activation globale, statut violation et IRQ.
0x010	EAR0	Adresse fautive capturée (port 0).
0x014	EDR0	Détails de violation (droits, maître, type d'accès).
0x400	RGD0_WORD0	Adresse de début de région 0.
0x404	RGD0_WORD1	Adresse de fin ou masque région 0 (selon variante).
0x408	RGD0_WORD2	Permissions lecture/écriture/exécution par domaine.
0x40C	RGD0_WORD3	Attributs validité, lock et contrôle région.
0x800	RGD1_WORD0	Début région 1 (même schéma que région 0).
0x804	RGD1_WORD1	Fin ou masque région 1.
0x900	RGDAAC0	Contrôle d'accès alternatif par maître/bus.

Offsets relatifs au bloc SMPU ; nomenclature exacte selon sous-famille MPC57xx.

FIGURE 7.2 – Mapping typique des registres SMPU sur MPC57xx

7.2.1 Exemple d'implémentation assembleur SMPU (MPC57xx)

L'exemple ci-dessous illustre une initialisation minimale de deux régions : Flash en lecture+exécution et SRAM en lecture/écriture non exécutable, puis activation globale de la protection. Les offsets exacts et bits de permission varient selon sous-famille ; le schéma reste représentatif de la logique de programmation [19].

```
/* Exemple simplifie e200/MPC57xx - syntaxe GNU as */

.equ SMPU_BASE,         0xFC010000
.equ SMPU_CESR,         (SMPU_BASE + 0x000)
.equ SMPU_RGD0_W0,      (SMPU_BASE + 0x400)
.equ SMPU_RGD0_W1,      (SMPU_BASE + 0x404)
.equ SMPU_RGD0_W2,      (SMPU_BASE + 0x408)
```

```
.equ SMPU_RGD0_W3,      (SMPU_BASE + 0x40C)
.equ SMPU_RGD1_W0,      (SMPU_BASE + 0x800)
.equ SMPU_RGD1_W1,      (SMPU_BASE + 0x804)
.equ SMPU_RGD1_W2,      (SMPU_BASE + 0x808)
.equ SMPU_RGD1_W3,      (SMPU_BASE + 0x80C)

/* Constantes simplifiees de droits */
.equ CESR_ENABLE,       0x00000001
.equ RGD_VALID,         0x00000001
.equ RGD_PERM_RX,       0x0000002A   /* domaine: lecture+execution */
.equ RGD_PERM_RW,       0x00000033   /* domaine: lecture+ecriture */
.equ RGD_XN,            0x00000100   /* execute-never */

smpu_init_mpc57xx:
  /* 1) Desactiver SMPU pendant programmation */
  lis   r3, SMPU_CESR@h
  ori   r3, r3, SMPU_CESR@l
  li    r4, 0
  stw   r4, 0(r3)
  msync
  isync

  /* 2) Region 0: Flash 0x00400000..0x004FFFFF, RX */
  lis   r3, SMPU_RGD0_W0@h
  ori   r3, r3, SMPU_RGD0_W0@l
  lis   r4, 0x0040
  stw   r4, 0(r3)

  lis   r3, SMPU_RGD0_W1@h
  ori   r3, r3, SMPU_RGD0_W1@l
  lis   r4, 0x004F
  ori   r4, r4, 0xFFFF
  stw   r4, 0(r3)

  lis   r3, SMPU_RGD0_W2@h
  ori   r3, r3, SMPU_RGD0_W2@l
  li    r4, RGD_PERM_RX
  stw   r4, 0(r3)

  lis   r3, SMPU_RGD0_W3@h
  ori   r3, r3, SMPU_RGD0_W3@l
  li    r4, RGD_VALID
  stw   r4, 0(r3)

  /* 3) Region 1: SRAM 0x40000000..0x4001FFFF, RW + XN */
  lis   r3, SMPU_RGD1_W0@h
  ori   r3, r3, SMPU_RGD1_W0@l
  lis   r4, 0x4000
  stw   r4, 0(r3)

  lis   r3, SMPU_RGD1_W1@h
  ori   r3, r3, SMPU_RGD1_W1@l
```

```
60    lis   r4, 0x4001
61    ori   r4, r4, 0xFFFF
62    stw   r4, 0(r3)
63
64    lis   r3, SMPU_RGD1_W2@h
65    ori   r3, r3, SMPU_RGD1_W2@l
66    li    r4, (RGD_PERM_RW | RGD_XN)
67    stw   r4, 0(r3)
68
69    lis   r3, SMPU_RGD1_W3@h
70    ori   r3, r3, SMPU_RGD1_W3@l
71    li    r4, RGD_VALID
72    stw   r4, 0(r3)
73
74    /* 4) Reactiver SMPU */
75    lis   r3, SMPU_CESR@h
76    ori   r3, r3, SMPU_CESR@l
77    li    r4, CESR_ENABLE
78    stw   r4, 0(r3)
79    msync
80    isync
81    blr
```

7.3 Variantes selon sous-famille

La famille MPC57xx couvre des profils différents (body/chassis, powertrain, gateway) et des écarts d'intégration qui impactent directement la politique mémoire :

- nombre d'entrées MPU coeur (TLB statiques) et granularité effective par coeur e200 ;
- support des registres MAS étendus (ex. MAS7/MAS6) selon coeur et espace adressable ;
- nombre de régions et granularité effectivement disponibles ;
- nombre de ports/maîtres supervisés ;
- sémantique précise des mots de région (W0..W3) et des bits de lock ;
- comportement des exceptions (Data/Instruction storage) et registres de diagnostic associés.

En pratique, la migration entre variantes MPC574x et MPC577x ne doit pas se limiter à un recopiage d'offsets. Il faut revalider séparément la couverture MPU coeur (zones code/données par contexte d'exécution) et la couverture SMPU système (maîtres de bus, DMA, périphériques), ainsi que les anti-patterns de recouvrement involontaire [19, 20].

8

PMP sur RISC-V : fonctionnement et variantes

8.1 Logique générique RISC-V (PMP)

Sur RISC-V embarqué, le mécanisme équivalent à la MPU est la PMP (Physical Memory Protection). La PMP est programmée via des CSRs dédiés (`pmpcfgX`, `pmpaddrX`) et applique des droits R/W/X sur des intervalles physiques, avec des modes de matching TOR, NA4 ou NAPOT selon le format retenu [24, 25].

Les principes opérationnels sont proches d'ARM/PowerPC :

1. une entrée PMP définit une zone physique ;
2. un champ de configuration fixe droits et mode d'adressage ;
3. la priorité est donnée à l'entrée de plus petit index qui matche ;
4. une violation déclenche une exception de type instruction/load/store access fault.

La PMP a toutefois deux particularités structurantes pour la sécurité :

- le bit `L` (lock) peut figer une entrée jusqu'au reset ;
- selon l'implémentation, les accès en mode Machine (M-mode) peuvent être soumis ou non à certaines règles PMP, ce qui impose de clarifier le modèle privilège dans la stratégie de défense.

8.1.1 Exemple d'implémentation assembleur RISC-V

L'exemple suivant montre une configuration PMP minimale avec deux entrées TOR : région Flash execute/read-only puis région SRAM read/write no-exec. Les encodages exacts dépendent de l'architecture (RV32/RV64) et du nombre d'entrées implémentées sur le coeur cible [24, 25].

```
/* Exemple simplifie RISC-V PMP - syntaxe GNU as */

.equ PMP_R,       0x01
.equ PMP_W,       0x02
.equ PMP_X,       0x04
.equ PMP_A_TOR,   0x08
```

CSR	Nom	Rôle principal
0x300	mstatus	Contexte privilège ; interaction avec la délégation des traps.
0x305	mtvec	Vecteur des exceptions (incluant faults PMP).
0x340	mscratch	Sauvegarde de contexte minimale en handler machine.
0x342	mcause	Cause de faute (load/store/instruction access fault).
0x343	mtval	Adresse fautive en cas de violation PMP.
0x3A0	pmpcfg0	8 champs de configuration PMP (R/W/X/A/L).
0x3A1	pmpcfg1	Entrées PMP supplémentaires (si implémentées).
0x3B0	pmpaddr0	Adresse associée à l'entrée PMP 0.
0x3B1	pmpaddr1	Adresse associée à l'entrée PMP 1.
0x3BF	pmpaddr15	Dernière entrée typique (nombre variable selon coeur).

Les CSRs PMP disponibles dépendent de l'implémentation (E31, U54, etc.).

FIGURE 8.1 – Mapping typique des CSRs liés à la PMP en RISC-V

```
.equ PMP_L,       0x80

/* pmpcfg0 octet0 -> entree 0 ; octet1 -> entree 1 */

mpu_init_riscv_pmp:
  /* 1) Nettoyage config PMP */
  li      t0, 0
  csrw    pmpcfg0, t0

  /* 2) Entree 0 TOR: [0x00000000, 0x080FFFFF] RX */
  li      t0, (0x080FFFFF >> 2)
  csrw    pmpaddr0, t0

  li      t1, (PMP_R | PMP_X | PMP_A_TOR)

  /* 3) Entree 1 TOR: (0x080FFFFF, 0x2001FFFF] RW, no-exec */
  li      t0, (0x2001FFFF >> 2)
  csrw    pmpaddr1, t0

  li      t2, (PMP_R | PMP_W | PMP_A_TOR)

  /* 4) Composer pmpcfg0: cfg1 dans l'octet 1, cfg0 dans l'octet 0 */
  slli    t2, t2, 8
  or      t3, t2, t1
  csrw    pmpcfg0, t3

  fence
  ret
```

8.2 Variantes selon implémentations

La portabilité PMP est moins immédiate qu'il n'y paraît, car la spécification autorise des choix d'implémentation importants :

- nombre d'entrées (4, 8, 16, 64) ;
- présence/absence de mode S/U et délégations d'exceptions ;
- comportement précis des accès M-mode vis-à-vis de PMP ;
- support de Smepmp (extensions de renforcement sur cibles récentes).

Il faut donc documenter explicitement le *security contract* de la plateforme : quels modes exécutent l'application, quelles zones sont lockées, et quels handlers traitent les access faults avec quelle politique fail-safe.

9
Comparatif sécurité entre architectures

9.1 Correspondances de concepts

Le tableau 9.1 aligne les concepts de protection mémoire utiles pour la sécurité entre ARM Cortex-M, PowerPC MPC57xx et RISC-V PMP.

Concept	**ARM Cortex-M**	**PowerPC MPC57xx**	**RISC-V PMP**
Unité de protection	MPU (PMSAv7/v8)	SMPU / MPU système	PMP (CSRs)
Définition région	Base+taille (v7) ou base+limite (v8)	Début/fin + droits par région	TOR/NA4/NAPOT via pmpaddr/pmpcfg
Priorité recouvrement	Numéro de région le plus élevé	Règles par région/port selon implémentation	Plus petit index PMP qui matche
Exécution sur données	Bit XN / Execute-Never	Attribut execute contrôlé par région	Absence de X sur entrée PMP
Modes privilège	Priv/Unpriv (+ Secure/NS en v8-M TZ)	Supervisor/User (MSR[PR])	M/S/U selon coeur
Diagnostics faute	MemManage + CFSR/MMFAR	Exceptions storage + ESR/DEAR	mcause/mtval access fault
Gouvernance DMA	Souvent externe à MPU CPU	Plus naturellement intégrée multi-maîtres	Dépend du sous-système bus/IOMMU local

Table 9.1 – Comparatif des concepts de protection mémoire selon architecture

9.2 Impact sur la portabilité

La portabilité d'une politique de sécurité mémoire n'est pas un exercice de traduction registre-à-registre ; c'est une conservation de propriétés de sécurité.

En pratique, il faut maintenir au minimum les invariants suivants lors d'une migration ARM → MPC57xx → RISC-V (ou inverse) :

1. **Code en RX uniquement** : aucune zone de données ne doit rester exécutable.
2. **Données critiques non modifiables** depuis les domaines applicatifs.
3. **MMIO sensibles réservés** au contexte privilégié.
4. **Gestion déterministe des fautes** avec journalisation et réaction fail-safe.
5. **Contrôles DMA cohérents** avec la politique CPU.

Les écarts les plus fréquents observés en migration sont :

- transposition naïve des tailles de région ARMv7 vers TOR/NAPOT en PMP ;
- oubli de la priorité d'index PMP (effets de masquage de règles) ;
- hypothèse erronée que la protection CPU couvre automatiquement tous les maîtres de bus ;
- non prise en compte des comportements lock/défaut selon architecture.

Une migration robuste impose donc une double vérification : preuve statique de couverture des zones critiques et tests négatifs dynamiques sur chaque architecture cible [5, 19, 24, 11].

10
La MPU comme contre-mesure aux attaques mémoire

Ce chapitre consolide les éléments du modèle de menace (chapitre 4) et des objectifs de sécurité OS-1 à OS-5 (section 4.2) pour expliciter, attaque par attaque, la contribution réelle de la MPU. L'approche retenue est opérationnelle : pour chaque scénario, on décrit la chaîne d'exploitation, l'impact attendu sans protection, puis le rempart apporté par une politique MPU correctement implémentée.

10.1 Principes directeurs de la contre-mesure MPU

La MPU n'est efficace en cybersécurité que si la politique mémoire respecte simultanément quatre principes d'ingénierie :

1. **Deny-by-default** : toute zone non explicitement décrite reste inaccessible.
2. **Moindre privilège** : chaque composant n'obtient que les droits strictement nécessaires (R/W/X et niveau de privilège).
3. **Séparation code/données** : le code est exécutable et non modifiable ; les données sont non exécutables.
4. **Cloisonnement par domaine de confiance** : noyau, drivers, application, diagnostic et mise à jour sont isolés autant que la granularité matérielle le permet.

Ces principes structurent directement les objectifs OS-1 (confinement), OS-2 (intégrité du flux d'exécution), OS-3 (protection des structures critiques), OS-4 (restriction MMIO) et OS-5 (détection/réaction) définis plus haut.

10.2 Attaque 1 : corruption mémoire et détournement de flux

10.2.1 Scénario de menace

La chaîne d'attaque type est : entrée non fiable (réseau, bus, interface locale), corruption d'un buffer, altération d'une structure de contrôle (adresse de retour, pointeur de fonction, entrée de table), puis redirection du flux d'exécution [29]. Sur MCU sans isolation, cette chaîne mène rapidement à une compromission globale, comme le montrent les analyses de binaires bare-metal par Clements *et al.* [11, 10].

Du point de vue cas concret, la compromission distante d'ECU illustrée par Checkoway *et al.* et Miller/Valasek montre qu'un vecteur d'entrée externe peut être converti en contrôle interne de fonctions critiques quand les frontières mémoire et privilèges sont insuffisantes [9, 18].

10.2.2 Rempart MPU

La MPU agit à deux niveaux de la chaîne d'exploitation :

- **Confinement de la corruption (OS-1, OS-3).** La RAM est segmentée en régions par domaine (pile tâche, données applicatives, structures noyau/RTOS, vecteurs d'interruptions). Une écriture hors domaine déclenche une faute au lieu d'une corruption silencieuse.
- **Réduction de l'impact (OS-5).** Le gestionnaire de faute traite l'événement de manière déterministe (journalisation + transition vers état sûr), ce qui limite la propagation de l'incident.

En pratique, le couple *partitionnement strict + fautes bloquantes* transforme une compromission potentiellement systémique en faute localisée et observable.

10.3 Attaque 2 : exécution de code non autorisé

10.3.1 Scénario de menace

Après corruption mémoire, l'attaquant cherche soit à exécuter un payload injecté en RAM (shellcode), soit à chaîner des gadgets déjà présents dans le binaire légitime (ROP/JOP) [29]. Sans séparation stricte code/données, la RAM devient une surface d'exécution triviale.

Dans les attaques industrielles documentées (ex. Stuxnet, Triton/TRISIS), l'objectif opérationnel est de faire exécuter un comportement non autorisé sur des couches basses du système, avec persistance et discrétion [14, 12]. Même si ces campagnes ne se réduisent pas à la MPU, elles illustrent l'impact d'une exécution non maîtrisée sur un système de contrôle.

10.3.2 Rempart MPU

La contribution centrale de la MPU est la fermeture des surfaces exécutables :

- **XN systématique sur données (OS-2).** Piles, tas, buffers I/O et zones partagées sont marqués non exécutables.

- **Code en RX uniquement (OS-2).** Les régions Flash/code restent exécutables mais non modifiables en runtime.
- **Réduction de la matière exploitable pour ROP/JOP.** En minimisant les régions exécutables et en évitant les mappings permissifs, on augmente les prérequis techniques de l'attaquant et on réduit les chaînes exploitables.

La MPU n'élimine pas toutes les formes de ROP/JOP, mais elle retire le chemin d'injection directe de code en RAM et contribue fortement à l'élévation du coût d'attaque.

10.4 Attaque 3 : élévation de privilège

10.4.1 Scénario de menace

Un code déjà exécuté en contexte non privilégié tente d'accéder aux ressources noyau : structures d'ordonnancement RTOS, table des vecteurs, registres MMIO sensibles (DMA, contrôles Flash, debug), ou mécanismes de reconfiguration mémoire. Sans séparation privilège stricte, une compromission locale devient une prise de contrôle globale.

Les travaux sur le cloisonnement embarqué montrent que la frontière user/superviseur est le facteur déterminant entre incident local et escalade noyau [11, 10].

10.4.2 Rempart MPU

La MPU fournit le rempart d'escalade via trois mécanismes complémentaires :

- **Séparation privilégié/non privilégié (OS-3, OS-4).** Les structures noyau et MMIO critiques sont accessibles uniquement depuis le contexte privilégié.
- **Passerelle d'accès contrôlée.** Les tâches applicatives passent par des appels système explicites (SVC/syscall) ; les paramètres sont validés côté noyau avant toute opération sensible.
- **Protection de la politique elle-même.** Les registres MPU et chemins de reconfiguration restent sous contrôle privilégié, limitant la possibilité de désactiver la barrière depuis un contexte compromis.

Cette stratégie matérialise la frontière de confiance : un code applicatif compromis peut échouer localement sans obtenir l'accès aux actifs souverains du système.

10.5 Synthèse opérationnelle : menace, objectif, rempart MPU

Le tableau 10.1 synthétise la correspondance entre menaces, objectifs et effets attendus de la contre-mesure MPU.

Scénario d'attaque	OS visé	Rempart MPU principal
Corruption mémoire (stack/heap/pointeurs)	OS-1, OS-3, OS-5	Cloisonnement RAM par domaine, zones critiques en écriture restreinte, faute matérielle et traitement fail-safe.
Exécution non autorisée (shellcode RAM, prérequis ROP/JOP)	OS-2, OS-5	XN sur toutes les données, code en RX uniquement, minimisation des régions exécutables et gestion de faute déterministe.
Élévation de privilège depuis tâche compromise	OS-3, OS-4, OS-5	Séparation privilège stricte, MMIO sensibles réservés au noyau, accès via syscall contrôlé, protection de la configuration MPU.

TABLE 10.1 – Contribution de la MPU comme rempart face aux scénarios d'attaque mémoire

10.6 Conditions de validité de la contre-mesure

Pour que ce rempart soit effectif dans un produit réel, cinq conditions de déploiement restent non négociables :

1. activation précoce et vérifiée de la MPU au démarrage ;
2. politique par défaut restrictive sur les zones non mappées ;
3. absence de recouvrement ambigu ou permissif des régions ;
4. gestionnaire de faute testé en campagne négative ;
5. cohérence avec les protections non-MPU (secure boot, verrouillage debug, gouvernance DMA).

En synthèse, la MPU n'est pas une protection universelle, mais une barrière structurelle de premier rang contre les chaînes d'attaque mémoire : elle réduit l'exploitabilité, limite l'impact des compromissions locales et fournit un mécanisme de détection/réaction indispensable dans une architecture de défense en profondeur.

11

MPU face aux attaques DMA et périphériques compromis

11.1 Risques spécifiques DMA

Le DMA introduit un changement de modèle fondamental : les transferts mémoire ne sont plus uniquement initiés par le coeur CPU, mais aussi par des maîtres de bus autonomes. Une politique MPU processeur peut donc être parfaitement correcte tout en restant contournable par un DMA mal configuré.

Les scénarios de risque principaux sont :

- **lecture de zones sensibles** (clés, état noyau, buffers inter-domaines) via une fenêtre DMA trop large ;
- **écriture non autorisée** dans des structures de contrôle (vecteurs, descripteurs, TCB RTOS) ;
- **pivot d'attaque** depuis un périphérique exposé (réseau, radio, stockage) vers la RAM système ;
- **contournement de logique applicative** en modifiant des buffers après validation logicielle.

Ce risque est particulièrement critique sur plateformes multi-maîtres (automobile/industriel) où le bus interconnecte CPU, DMA, accélérateurs et contrôleurs de communication [19, 21].

11.2 Mesures défensives

Une stratégie robuste combine configuration MPU CPU et gouvernance bus :

1. **Fenêtres DMA minimales.** Chaque canal DMA est borné au strict sous-ensemble mémoire nécessaire.
2. **Segmentation des tampons.** Séparer buffers DMA, buffers applicatifs et données critiques en régions distinctes.

3. **Protection système multi-maîtres.** Utiliser XRDC/SMPU/IOMMU selon architecture pour imposer des droits par maître.
4. **MMIO en liste blanche.** Restreindre la reconfiguration des contrôleurs DMA au code privilégié.
5. **Contrôles temporels.** Désactiver les canaux DMA hors phase d'usage et invalider les descripteurs après transfert.

La MPU reste un rempart utile, mais son rôle face au DMA est indirect : elle protège le chemin CPU et réduit les possibilités d'escalade après compromission, tandis que le confinement inter-maîtres doit être traité au niveau du sous-système bus.

12
Erreurs de configuration fréquentes et anti-patterns

12.1 Défauts de design

Les défauts de conception les plus fréquents observés en revue sont les suivants :

- **Régions trop larges.** Une seule région RAM RWX couvre plusieurs domaines de confiance et annule le cloisonnement.
- **Recouvrements non maîtrisés.** Une région prioritaire plus permissive masque une région censée être restrictive.
- **Politique de fond permissive.** Le *background map* autorise implicitement des accès non prévus.
- **Attributs incohérents.** Données marquées exécutables, code marqué writable, MMIO exposé en non privilégié.
- **Absence de modèle d'allocation des régions.** Les régions sont choisies par opportunité, sans matrice explicite actif/domaine/droit.

Pour éviter ces anti-patterns, la configuration doit être dérivée d'exigences traçables (OS-1 à OS-5), puis vérifiée par inspection croisée (code, linker script, documentation sécurité).

12.2 Défauts d'exploitation

Même avec une bonne conception initiale, la protection dérive souvent en phase d'intégration :

- **Handlers de faute incomplets.** Journalisation absente, diagnostics insuffisants, redémarrage non déterministe.
- **Reconfiguration dynamique non contrôlée.** Changement de régions sans validation ni barrières mémoire.
- **Bypass en maintenance.** Activation temporaire de droits larges conservée en production.
- **Non-régression absente.** Une évolution applicative modifie la carte mémoire sans mise à jour des tests négatifs.
- **Incohérence multicore.** Politique appliquée sur un coeur mais pas sur les autres.

Le critère clé d'exploitation est simple : toute violation de politique doit être détectée, corrélée à une cause, et traitée selon une réaction fail-safe validée.

13
Intégration dans un cycle de développement sécurisé

13.1 Exigences et traçabilité

La sécurité MPU doit être gouvernée par une chaîne de traçabilité explicite :

1. **Exigence sécurité** : formulation de l'objectif (ex. OS-2, aucune exécution en RAM).
2. **Décision d'architecture** : définition du partitionnement et des privilèges.
3. **Implémentation** : configuration registres, linker script, wrappers syscalls.
4. **Preuve** : test positif/négatif, revue de code, preuve d'intégration.
5. **Critère d'acceptation** : décision de conformité ou dérogation motivée.

Un artefact type de traçabilité associe chaque région mémoire à : actif protégé, domaine autorisé, droits R/W/X, niveau de privilège, justification, test associé et résultat de validation.

13.2 Industrialisation

L'industrialisation de la politique MPU repose sur quatre pratiques :

- **Revues systématiques.** Revue sécurité dédiée à chaque évolution de carte mémoire.
- **Validation automatisée.** Intégration des campagnes de tests MPU en CI/CD (émulation, bench, HIL selon disponibilité).
- **Diff de configuration lisible.** Générer des rapports de différences sur les régions et permissions entre versions.
- **Gouvernance release.** Blocage de version en cas d'échec des tests négatifs ou de dérive des droits.

Cette discipline réduit les régressions silencieuses, qui sont la première cause de perte d'efficacité des mécanismes de cloisonnement mémoire en exploitation.

14
Validation et tests de robustesse

14.1 Tests de sécurité

La validation doit combiner tests fonctionnels et tests d'abus orientés attaque :

- **Tests positifs.** Vérifier que chaque domaine accède uniquement aux zones autorisées.
- **Tests négatifs lecture/écriture/exécution.** Forcer des accès interdits pour déclencher les fautes attendues.
- **Tests de transition de contexte.** Vérifier la reconfiguration MPU à chaque changement de tâche/mode.
- **Tests MMIO/DMA.** Tenter des accès non autorisés aux périphériques et canaux DMA.
- **Campagnes de robustesse.** Fault injection logicielle et perturbations contrôlées pour valider le comportement fail-safe.

Chaque test négatif doit vérifier trois résultats simultanés : faute matérielle correcte, journalisation exploitable, réaction conforme à la politique de sûreté/cybersécurité.

14.1.1 Exemple de suite de tests de conformité complète

Le tableau 14.1 propose une suite de tests type assurant une couverture complète de la conformité de l'implémentation MPU aux objectifs OS-1 à OS-5. Chaque test doit être exécuté au minimum en environnement de qualification (banc/HIL), et rejoué en non-régression à chaque évolution de cartographie mémoire.

En complément, il est recommandé d'associer à chaque test un identifiant unique, une précondition d'exécution, un oracle de verdict, et une preuve archivée (log brut, trace horodatée, dump de registres de faute) pour audit.

14.2 Critères d'acceptation

Les critères d'acceptation minimaux recommandés sont :

1. **Couverture des actifs critiques.** 100% des actifs classés critiques sont associés à une règle MPU explicite.
2. **Couverture des interdictions.** 100% des interdictions de la matrice sécurité ont un test négatif automatisé.
3. **Reproductibilité.** Résultats stables sur plusieurs builds et plusieurs cibles matérielles.
4. **Temps de réaction.** Comportement de faute borné et compatible avec les contraintes système.
5. **Absence de contournement connu.** Aucun chemin documenté ne permet l'accès hors politique sans alerte.

Les écarts résiduels doivent être traités par dérogation formelle avec analyse d'impact, plan d'action et date de clôture.

Intitulé du test	Description	OS associé
Confinement inter-domaines en écriture	Depuis une tâche applicative non privilégiée, tenter d'écrire dans la RAM d'un autre domaine (ou dans une zone noyau). Résultat attendu : faute MPU, absence de modification mémoire cible, journalisation de l'adresse fautive.	OS-1
Confinement inter-domaines en lecture	Depuis un domaine applicatif, tenter de lire une zone sensible d'un autre domaine (clé, état RTOS, configuration critique). Résultat attendu : accès refusé et traçabilité de l'événement.	OS-1
Exécution interdite depuis la RAM (XN)	Injecter un stub d'instructions dans pile/heap/buffer puis forcer un saut vers cette zone. Résultat attendu : exception d'exécution, aucun fetch valide en RAM.	OS-2
Intégrité des zones code (Flash en RX)	Tenter une écriture applicative dans une région code marquée RX. Résultat attendu : écriture rejetée, contenu inchangé (hash/CRC identique).	OS-2
Protection de la table des vecteurs	En contexte non privilégié, tenter de modifier une entrée de table des vecteurs. Résultat attendu : écriture refusée et vecteur inchangé après vérification.	OS-3
Protection des structures RTOS critiques	Simuler une altération de TCB/listes d'ordonnancement depuis une tâche non autorisée. Résultat attendu : refus matériel et absence de corruption des structures noyau.	OS-3
Restriction MMIO sensible	Depuis une tâche non privilégiée, tenter l'accès à des registres MMIO sensibles (DMA, flash ctrl, debug). Résultat attendu : accès interdit ; seuls les services noyau autorisés réussissent.	OS-4
Validation des passerelles syscall	Exécuter des appels système légitimes et malformés (pointeurs hors zone, tailles invalides). Résultat attendu : acceptation stricte des cas valides, rejet des cas invalides sans élargir les droits.	OS-3, OS-4
Réaction déterministe aux fautes MPU	Provoquer des violations R/W/X contrôlées et vérifier la séquence complète (capture contexte, code erreur, action fail-safe, redémarrage si requis).	OS-5
Non-régression de configuration MPU	Comparer deux versions firmware : toute différence de régions/permissions doit être détectée, justifiée et couverte par tests mis à jour.	OS-1 à OS-5

TABLE 14.1 – Exemple de suite de tests couvrant la conformité de l'implémentation MPU aux objectifs OS-1 à OS-5

15
Limites de la MPU et défenses complémentaires

15.1 Limites structurelles

Même correctement configurée, la MPU conserve des limites intrinsèques :

- **Granularité régionale.** Les attaques intra-région restent possibles.
- **Absence de validation logique.** Un code autorisé peut rester dangereux sur le plan fonctionnel.
- **Contournement par maîtres externes.** DMA et bus masters nécessitent des protections dédiées.
- **Hors périmètre physique.** Glitching, side-channels et attaques invasives ne sont pas couverts.
- **Fenêtre pré-activation.** Le code exécuté avant activation MPU reste non contraint.

La MPU doit donc être considérée comme contrôle nécessaire, mais non suffisant, d'une défense en profondeur.

15.2 Mesures complémentaires

Les compléments prioritaires sont :

- **Secure boot et chaîne de mise à jour authentifiée.**
- **Verrouillage des interfaces debug** (JTAG/SWD/UART maintenance).
- **Protection de secrets** (stockage sécurisé, gestion des clés, dérivation en runtime).
- **Supervision runtime** (watchdog, contrôles d'intégrité, télémétrie de faute).
- **Durcissement logiciel** (analyses statiques, revues, règles de codage et fuzzing ciblé).

L'efficacité réelle repose sur l'orchestration cohérente de ces contrôles, et non sur un mécanisme unique.

16
Migration et portabilité des politiques MPU

16.1 Méthodologie de translation

Une migration robuste ne traduit pas des registres, elle conserve des propriétés de sécurité. La méthode recommandée est :

1. **Formaliser les invariants** (code RX, données XN, MMIO privilégié, etc.).
2. **Projeter les actifs** sur la carte mémoire de la cible d'arrivée.
3. **Mapper les mécanismes** (PMSAv7/v8, SMPU, PMP TOR/NAPOT) sans hypothèse implicite.
4. **Vérifier les priorités/résolutions** des recouvrements propres à l'architecture.
5. **Rejouer la suite de tests négatifs** et comparer les résultats sécurité attendus.

La preuve de migration doit démontrer l'équivalence de sécurité, pas seulement la validité fonctionnelle.

16.2 Pièges fréquents

Les échecs de portabilité les plus fréquents sont :

- transposition directe des tailles/alignements ARMv7 vers PMP TOR/NAPOT ;
- oubli des règles de priorité différentes selon architecture ;
- confusion entre protection CPU et protection système multi-maîtres ;
- non prise en compte des modes privilèges réellement utilisés en production ;
- absence de verrouillage (bit `L` PMP ou mécanisme équivalent) quand requis.

Une stratégie de migration mature inclut une matrice de correspondance par propriété (objectif, mécanisme source, mécanisme cible, test de preuve).

17
Annexes techniques

17.1 Ressources de référence

Les annexes servent de trousse opérationnelle pour les équipes architecture, développement, validation et audit.

17.1.1 Matrices et check-lists recommandées

Documents de travail à maintenir en configuration contrôlée :

- matrice actif × domaine × droits R/W/X ;
- table des régions (base, taille/limite, attributs, justification, propriétaire) ;
- check-list de revue MPU (design, implémentation, tests, exploitation) ;
- trame de test négatif par objectif OS-1 à OS-5 ;
- plan de non-régression sécurité et critères de blocage release.

17.2 Panorama de solutions supportant la MPU

Le tableau 17.1 présente un panorama de solutions représentatives (bare metal, RTOS et noyaux cloisonnants) pouvant implémenter une politique MPU alignée avec les objectifs OS-1 à OS-5.

Type	**Solution**	**Editeur**	**Open source**	**ARM Cortex-M**	**PowerPC MPC57xx**	**RISC-V PMP**	**Objectifs de securite couverts**
Bare metal	CMSIS-Core + politique MPU applicative	Arm Ltd. (CMSIS) + equipe produit	Oui (CMSIS)	Oui	Non	Non	OS-2, OS-3, OS-4, OS-5 ; OS-1 partiel (selon partitionnement applicatif)
Bare metal	S32 SDK / drivers MPU-SMPU	NXP	Partiel (mixte)	Oui (S32K3)	Oui (MPC57xx)	Non	OS-1 a OS-5 (avec configuration SMPU + gestion fautes)
RTOS	FreeRTOS-MPU	Amazon / FreeRTOS community	Oui (MIT)	Oui	Non	Oui	OS-1 a OS-5 cote CPU ; OS-4 complet si MMIO filtre via services privilegies
RTOS	Zephyr (Userspace + Memory Domains)	Linux Foundation	Oui (Apache-2.0)	Oui	Partiel	Oui	OS-1 a OS-5 (selon activation userspace, domains et handlers)
RTOS	AUTOSAR Classic OS (ex. RTA-OS, MICROSAR OS)	ETAS / Vector (selon stack)	Non	Oui	Oui	Non	OS-1, OS-3, OS-4, OS-5 ; OS-2 selon integration XN/W^X et chaine boot
Noyau cloisonnant	Tock OS (capsules + processus isoles)	Tock Project	Oui (Apache-2.0/MIT)	Oui	Non	Oui	OS-1 a OS-5 sur cible compatible MPU/PMP
Noyau cloisonnant	seL4 (microkernel)	seL4 Foundation / UNSW	Oui (BSD)	Oui	Non	Oui	OS-1 a OS-5 (avec politique de capabilites et partitionnement configure)
Noyau cloisonnant	PikeOS	SYSGO	Non	Oui	Oui	Partiel	OS-1 a OS-5 (selon profil certifie, BSP et partitionnement actif)

TABLE 17.1 – Panorama de solutions supportant la MPU sur les trois architectures et couverture des objectifs de securite

Note : la couverture indiquee correspond a un potentiel de conformite, conditionne par la configuration effective (partitionnement, droits, handlers, restrictions MMIO/DMA) et par le niveau de maturite du BSP sur la cible.

17.2.1 Glossaire et acronymes

API. *Application Programming Interface* : interface logicielle exposée à un composant externe.

ARM. Famille d'architectures processeur largement utilisée dans les MCU et SoC embarqués.

CAN. *Controller Area Network* : bus de communication série largement utilisé en automobile et industriel.

CC. *Common Criteria* : cadre d'évaluation de sécurité de produits informatiques.

CI/CD. *Continuous Integration / Continuous Delivery* : automatisation de l'intégration, des tests et des livraisons.

CFSR. *Configurable Fault Status Register* : registre Cortex-M de diagnostic des fautes configurables.

CMSIS. *Cortex Microcontroller Software Interface Standard* : couche standard ARM pour coeurs Cortex-M.

CPU. *Central Processing Unit* : unité centrale de traitement exécutant le code machine.

CSR. *Control and Status Register* : registre de contrôle/état (notamment en RISC-V).

DMA. *Direct Memory Access* : transfert mémoire sans intervention directe du CPU.

ECU. *Electronic Control Unit* : unité de contrôle électronique, notamment en automobile.

EAL. *Evaluation Assurance Level* : niveau d'assurance dans Common Criteria.

ESR. *Exception Syndrome Register* : registre d'état d'exception (PowerPC e200).

Flash. Mémoire non volatile contenant généralement le firmware exécutable.

GNU. *GNU's Not Unix* : projet libre dont les outils (par ex. assembleur GNU as) sont utilisés en embarqué.

HardFault. Exception Cortex-M de faute critique non récupérée par les handlers dédiés.

HIL. *Hardware-in-the-Loop* : méthode de test intégrant matériel réel et simulation.

IOMMU. *Input/Output Memory Management Unit* : unité de protection/traduction pour périphériques d'E/S.

ISA. *Instruction Set Architecture* : architecture du jeu d'instructions.

IVOR. *Interrupt Vector Offset Register* : registre de vecteur d'exception sur PowerPC e200.

JOP. *Jump-Oriented Programming* : technique d'exploitation par chaînage de gadgets de saut.

JTAG. *Joint Test Action Group* : interface standard de test et debug matériel.

LIN. *Local Interconnect Network* : bus série automobile bas coût.

MAIR. *Memory Attribute Indirection Register* : registre ARMv8-M d'indexation des attributs mémoire.

MCU. *Microcontroller Unit* : microcontrôleur.

MMFAR. *MemManage Fault Address Register* : registre Cortex-M contenant l'adresse fautive MemManage.

MMIO. *Memory-Mapped I/O* : registres de périphériques adressés comme de la mémoire.

MMU. *Memory Management Unit* : unité de gestion mémoire avec traduction d'adresses.

MPU. *Memory Protection Unit* : unité de protection mémoire par régions.

MSR. *Machine State Register* : registre d'état machine sur PowerPC.

NA4. *Naturally Aligned 4-byte region* : mode PMP pour région alignée de 4 octets.

NAPOT. *Naturally Aligned Power-Of-Two* : mode PMP pour région alignée de taille puissance de deux.

NVIC. *Nested Vectored Interrupt Controller* : contrôleur d'interruptions Cortex-M.

OS. Objectif de sécurité utilisé dans ce guide (OS-1 à OS-5).

OTA. *Over-The-Air* : mise à jour logicielle à distance.

PMP. *Physical Memory Protection* : mécanisme RISC-V de protection mémoire.

PMSA. *Protected Memory System Architecture* : modèle MPU des Cortex-M.

PMSAv7/PMSAv8. Versions du modèle PMSA correspondant aux générations ARMv7-M et ARMv8-M.

RAM. *Random Access Memory* : mémoire volatile de travail.

RBAR/RASR/RLAR. Registres ARM de description des régions MPU (base, attributs, limite).

RDP. *Readout Protection* : mécanisme de protection contre la lecture de la mémoire embarquée.

RO. *Read Only* : autorisation lecture seule.

ROM. *Read-Only Memory* : mémoire non volatile en lecture seule, souvent utilisée pour du code de démarrage.

ROP. *Return-Oriented Programming* : technique d'exploitation par chaînage de retours.

RTOS. *Real-Time Operating System* : système d'exploitation temps réel.

RW. *Read Write* : autorisation lecture et écriture.

RX. *Read Execute* : autorisation lecture et exécution.

SMPU. *System Memory Protection Unit* : protection mémoire système multi-maîtres.

Smepmp. Extension RISC-V de renforcement de la PMP pour modes privilégiés.

SoC. *System on Chip* : circuit intégrant plusieurs sous-systèmes sur une puce.

SVC. *Supervisor Call* : exception d'appel système sur ARM Cortex-M.

SWD. *Serial Wire Debug* : interface de debug ARM à deux fils.

TCB. *Task Control Block* : structure de contrôle d'une tâche RTOS.

TCP. *Transmission Control Protocol* : protocole de transport orienté connexion sur IP.

TLB. *Translation Lookaside Buffer* : cache des traductions/protections mémoire.

TOR. *Top Of Range* : mode d'adressage PMP basé sur borne supérieure.

TrustZone-M. Extension ARMv8-M de partitionnement Secure/Non-Secure.

UART. *Universal Asynchronous Receiver-Transmitter* : interface série asynchrone.

USB. *Universal Serial Bus* : interface série standard de communication et d'alimentation.

WBWA. *Write-Back, Write-Allocate* : politique de cache mémoire.

XN. *Execute Never* : attribut interdisant l'exécution d'instructions.

XRDC. *eXtended Resource Domain Controller* : contrôleur d'isolation de ressources/mémoire multi-maîtres.

Bibliographie

[1] Anvil Secure. Glitching STM32 read out protection with voltage fault injection. Technical report, Anvil Secure, 2024.

[2] Arm Ltd. *ARMv7-M Architecture Reference Manual*, 2010. Revision E.b.

[3] Arm Ltd. Arm trustzone for cortex-m. Technical report, Arm Ltd., 2017. Technical overview white paper.

[4] Arm Ltd. *Cortex-M7 Processor Technical Reference Manual*, 2020. Latest revision.

[5] Arm Ltd. *Armv8-M Architecture Reference Manual*, 2021. Latest revision.

[6] Arm Ltd. *Cortex-M33 Processor Technical Reference Manual*, 2021. Latest revision.

[7] Arm Ltd. *CMSIS-Core (Cortex-M) Documentation*, 2024.

[8] Arm Ltd. *CMSIS-Core MPU Functions and Register Abstraction*, 2024. See CMSIS groups group_mpu_functions and group_mpu8_functions.

[9] Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, Stefan Savage, Karl Koscher, Alexei Czeskis, Franziska Roesner, and Tadayoshi Kohno. Comprehensive experimental analyses of automotive attack surfaces. In *Proceedings of the 20th USENIX Security Symposium*. USENIX Association, 2011.

[10] Abraham A. Clements, Naif Saleh Almakhdhub, Saurabh Bagchi, and Mathias Payer. ACES : Automatic compartments for embedded systems. In *Proceedings of the 27th USENIX Security Symposium*. USENIX Association, 2018.

[11] Abraham A. Clements, Naif Saleh Almakhdhub, Khaled S. Saab, Prashast Srivastava, Jinkyu Koo, Saurabh Bagchi, and Mathias Payer. Protecting bare-metal embedded systems with privilege overlays. In *Proceedings of the 38th IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2017.

[12] Dragos Inc. TRISIS : Targeting safety instrumented systems. Technical report, Dragos Inc., 2017. Threat intelligence report, https://dragos.com/blog/trisis/.

[13] ENISA. ENISA threat landscape 2023. Technical report, European Union Agency for Cybersecurity, 2023.

[14] Nicolas Falliere, Liam O Murchu, and Eric Chien. W32.Stuxnet dossier. Technical report, Symantec Corporation, 2011. Version 1.4, https://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/w32_stuxnet_dossier.pdf.

[15] International Electrotechnical Commission. IEC 62443 series : Security for industrial automation and control systems. International standard series, 2018. Multi-part standard series.

[16] Keen Security Lab. Experimental security research of Tesla autopilot. Technical report, Tencent Keen Security Lab, 2016. Publicly released research report, https://keenlab.tencent.com/en/.

[17] Microchip Technology Inc. *SAM E70/S70/V70/V71 Family Data Sheet*, 2023. Latest revision.

[18] Charlie Miller and Chris Valasek. Remote exploitation of an unaltered passenger vehicle. In *DEF CON 23*, 2015. White paper, http://illmatics.com/Remote%20Car%20Hacking.pdf.

[19] NXP Semiconductors. *MPC5748G Reference Manual*, 2020. Latest revision.

[20] NXP Semiconductors. *MPC5744P Data Sheet*, 2021. Latest revision.

[21] NXP Semiconductors. *S32K3xx Reference Manual*, 2024. Latest revision.

[22] Timo Obermaier and Stefan Tatschner. Shedding too much light on a microcontroller's firmware protection. In *11th USENIX Workshop on Offensive Technologies (WOOT 17)*. USENIX Association, 2017.

[23] Sandro Pinto and Nuno Santos. Demystifying Arm TrustZone : A comprehensive survey. *ACM Computing Surveys*, 51(6), 2019.

[24] RISC-V International. *The RISC-V Instruction Set Manual Volume II : Privileged Architecture*, 2024. Ratified specification.

[25] SiFive Inc. *SiFive E31 Core Complex Manual*, 2019. Includes PMP programming model.

[26] STMicroelectronics. *RM0090 Reference Manual : STM32F405/407, STM32F415/417, STM32F42x and STM32F43x Advanced Arm-based 32-bit MCUs*, 2023. Latest revision.

[27] STMicroelectronics. *RM0433 Reference Manual : STM32H743/753 and STM32H750 Value Line Advanced Arm-based 32-bit MCUs*, 2024. Latest revision.

[28] STMicroelectronics. *RM0456 Reference Manual : STM32U575/585 Arm-based 32-bit MCUs*, 2024. Latest revision.

[29] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. SoK : Eternal war in memory. In *Proceedings of the 34th IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2013.