Evaluation d'une solution d'isolation pour objets contraints

Nicolas Dejon, Chrystel Gaber, Gilles Grimaud

Compas'2022 : Parallélisme/ Architecture/ Système
MIS/UPJV - Amiens France, 5-8 juillet 2022

Evaluation d'une solution d'isolation 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
Dans cet article, nous présentons l'évaluation de Pip-MPU qui est une solution d'isolation qui cible les objets contraints et basée sur la Memory Protection Unit (MPU). Nous y décrivons notre banc de test et la manière dont l'évaluation a été conduite. Notre prototype de Pip-MPU prend moins de 10 Ko de Flash pour son code et 550 octets de RAM pour sa pile. Pip-MPU affiche un coût supplémentaire de 16% en terme de cycles mesurés ainsi que sur sa consommation énergétique, mais réduit la mémoire accessible d'une application jusqu'à 2% de son envergure initiale et ses opérations privilégiées de 100%.

Mots-clés : isolation, Pip, MPU, objets contraints, évaluation

1. Introduction

L'isolation entre composants au sein d'un même système est une propriété précieuse afin de séparer les fonctionnalités critiques et mieux se protéger des attaques à distance. Les machines grand public ont depuis longtemps proposé des mécanismes de sécurité qui ont prouvé leur efficacité, à l'instar de Pip [20], seL4 [22] ou mC2/CertiKOS [19] qui assurent l'isolation entre composants par l'utilisation de la Memory Management Unit (MMU).
Dans ce papier, nous nous intéressons à Pip-MPU, l'adaptation de Pip destinée aux objets contraints. Pip-MPU est une implémentation du *framework* présenté par *Dejon et al.* [17, 16] qui a été spécialisé pour être conforme à la politique de sécurité de Pip. Le *framework* est basé sur la Memory Protection Unit (MPU) afin de mettre en place des espaces mémoire isolés imbriqués avec un ancrage matériel. Nous explorons dans cet article les surcoûts imputés à Pip-MPU, soit le coût effectif d'une solution d'isolation basée sur MPU. Notre contribution est l'évaluation de Pip-MPU en terme de cycles CPU, de temps d'initialisation, d'empreinte mémoire et de consommation énergétique. Notre analyse comporte également des métriques de sécurité telles que l'accessibilité de régions mémoire et la mesure du nombre de cycles privilégiés. Les détails d'implémentation de Pip-MPU, la spécialisation du *framework* et la vérification formelle des propriétés de sécurité habituellement liées à Pip, ne sont pas couverts dans cet article.
Le reste du papier est structuré en deux parties : la Section 2 présente les points de différence essentiels entre MMU et MPU ainsi que le positionnement des travaux. La Section 3 présente la manière dont a été conduite l'évaluation ainsi que les résultats et discussions associées portant sur les métriques observées.

2. Contexte

Nous ciblons les objets contraints de Classe 2 et supérieure de la classification IETF [3] (>50 Ko RAM et >250 Ko Flash).

2.1. Memory Management Unit versus Memory Protection Unit

Pip-MPU repose essentiellement sur la Memory Protection Unit (MPU) et la séparation des privilèges. Comme la Memory Management Unit (MMU), la MPU contrôle l'accès à des régions mémoire et peut leur attribuer des permissions d'accès en lecture-écriture-exécution. En revanche, de nombeuses différences séparent MMU et MPU, notamment le fait que les régions MPU sont beaucoup moins nombreuses, typiquement entre 8 et 16 régions MPU contre des millions de pages MMU. Par ailleurs, ces pages sont souvent de taille fixe alors que les régions MPU sont de taille variable. Aussi, la MMU est de nos jours difficilement séparable du processeur et tout le temps activée alors que la MPU peut être désactivée ou même ne pas être intégrée. Néanmoins, un grand nombre de SOCs (*system on a chip*) sur le marché l'intègre [15].

2.2. Travaux connexes

La communauté scientifique a investi de nombreux efforts dans les architectures de sécurité basées sur MPU (ACES, MINION, TrustLite, EwoK, TockOS, OPEC [14, 21, 23, 12, 24, 30]). Cependant, elles considèrent toutes une isolation horizontale à un seul niveau là où Pip-MPU conçoit une hiérarchie de partionnement horizontale et verticale. De plus, certains systèmes négligent la compartimentation (ACES) comme solution aux contraintes imposées par la MPU (nombre de régions MPU, alignement des régions) alors que Pip-MPU ne remet jamais sa compartimentation en question, par ailleurs dynamique contrairement à d'autres systèmes (tous sont fixés à la compilation sauf TockOS). Nous pourrions citer d'autres approches d'isolation, dont des approches hybrides qui, contrairement à Pip-MPU, modifient le matériel comme TyTAN [13] basé sur Trustlite, SMART [27], Sancus [25], CheriRTOS [29] ou uniquement logicielles comme illustré dans Security microvisor [10] ou PISTIS [18] mais leur isolation reste limitée à deux espaces mémoire. Certains modules matériels autres que la MPU existent et sont parfois utilisés pour mettre en place des enclaves comme ARM TrustZone [1] pour les architectures ARM ou Intel SGX [4] et les Memory Protection Keys (MPK) [26] pour les machines Intel. Néanmoins, le nombre d'espaces isolés reste limité par rapport à Pip-MPU, et de plus, Pip-MPU peut être utilisé conjointement aux enclaves pour objets contraints.

3. Evaluation

Nous évaluons notre implémentation de Pip-MPU en faisant exécuter une application au sein de notre prototype de Pip-MPU sur un micro-contrôleur basé sur un processeur ARMv7 Cortex-M et en comparant l'exécution à une implémentation de référence sans Pip. L'objectif de cette évaluation est de répondre aux questions suivantes : 1) Est-ce que la solution est utilisable sur des objets contraints ? 2) Quels sont les surcoûts liés aux performances (cycles, consommation énergétique) et à la présence de la couche logicielle supplémentaire (taille, lignes de code, temps d'initialisation) pour quels gains de sécurité (mémoire accessible, opérations privilégiées) ?

3.1. Banc d'essai et expérience

Notre prototype tourne sur un kit de développement nRF52840 Nordic Semiconductor [5] (nRF52840-DK). La puce embarque un CPU ARM Cortex-M4 (architecture ARMv7-M) fonctionnant jusqu'à 64 MHZ et comprenant une mémoire Flash de 1 Mo et une mémoire RAM de 256 ko, ainsi que d'une MPU composée de 8 régions MPU.
Nous conduisons des analyses statiques et dynamiques sur 4 applications issues de la suite de tests d'évaluations *Embench IoT benchmark suite* [2] : aha-mont64, crc32, nsichneu, primecount.

TABLE 1 – Surcoût de Pip-MPU. Pour mesurer la taille de Pip-MPU et son empreinte mémoire, nous avons utilisé l'option de compilation `-Os`.

Empreinte mémoire en Flash (octets/SLOC de C)	
Taille totale de Pip-MPU	9544 / 4186
Empreinte mémoire en RAM (octets)	
Pile de Pip-MPU	516
Structures de métadonnées :	
- par partition	$640 + (\texttt{B} \bmod 8) \times 512$
- minimum par partition	1152
- maximum par partition	4736
Déploiement (#cycles)	
Initialisation de Pip-MPU	Partition racine : 99022 / Partition enfant : 165582

L'évaluation consiste en deux scénarios Pip, faisant tourner une application au sein 1) de la partition racine ou 2) d'une partition enfant. De façon identique à Pip, la partition racine est lancée par Pip-MPU à la fin de son initialisation et est l'unique partition au démarrage. La partition racine initialise et lance ensuite elle-même des partitions enfant. Ces partitions enfant peuvent également créer des partitions enfant, et ainsi de suite. Toutes les partitions s'exécutent dans l'espace utilisateur. Nous avons comparé chaque scénario à notre scénario de référence qui est l'exécution de la même application en mode privilégié, sans Pip et lancée après la même phase d'initialisation. L'application est régulièrement interrompue par une interruption SysTick toutes les 10 ms. L'interruption déclenche une routine qui rend la main immédiatement dans le cas de notre scénario de référence, ou alors traverse la routine d'interruption de Pip-MPU dans les scénarios Pip. Une expérience associe une application avec un scénario. Nous distinguons quatre phases dans l'expérience : la phase d'initialisation du système, la phase d'initialisation de l'application (en fonction des scénarios, mise en place de la partition racine et de la partition enfant ou rien), la phase de test comprenant des lancements successifs du programme principal, et la phase de fin qui arrête l'expérience et envoie les données à l'ordinateur pilotant l'évaluation. Une analyse post-mortem est réalisée sur les données collectées de toutes les expériences afin d'extraire les informations et produire des rapports de statistique.

3.2. Métriques et résultats

Nous avons écrit des scripts Python pour piloter la phase d'évaluation, inspirés par et adaptés selon les scripts et les outils proposés par Embench and BenchIoT [9]. Dans cette section, nous décrivons les métriques que nous avons surveillées et comment nous avons récolté les données. Les résultats finaux présentent le surcoût de Pip-MPU dans le Tableau 1, et le surcoût induit sur les performances de l'application et les gains de sécurité obtenus dans le Tableau 2.

Les lignes de code source (*SLOC, Source Lines of Code*) sont le nombre de lignes de code source C après suppression de tous les commentaires et de toutes les lignes vide de tous les fichiers source C par l'utilisation de l'option GCC `-fpreprocessed`. Les lignes comptabilisées incluent celles qui ne contiennent que des accolades, des variables globales ou encore les paramètres de fonction pouvant s'étaler sur plusieurs lignes (même si peu fréquent). Le Tableau 1 présente les SLOC et la taille (en octets) de Pip-MPU, qui comporte ses services principaux et ses routines d'interruption, ainsi que les fonctions bas-niveau d'accès au matériel.

L'utilisation de la pile est mesurée sur les pile principale (*main stack*) et utilisateur (*user stack*) en les marquant au préalable par une valeur prédéfinie. A la fin du test, la dernière position comprenant la valeur prédéfinie témoigne de l'usage. En plus des métadonnées requises pour

TABLE 2 – Comparaison des performances et de la sécurité par rapport au scénario de référence. L'application est soit exécutée au sein de la partition racine, soit dans la partition enfant.

Métriques	**Partition racine**	**Partition enfant**
Cycles		
Cycles :		
i) au total	+16.31%	+16.4%
ii) pendant le test	+16.29%	+16.36%
Ratio des cycles privilégiés sur le total des cycles :		
i) au total	-99.14%	-99.08%
ii) pendant le test	-99.13%	-99.08%
Consommation énergétique pendant la phase de test		
Totale	+16.7%	+18.4%
dont causée par la MPU	+0.03%	+0.04%
Securité		
Ratio de la mémoire accessible depuis l'application sur la mémoire totale :		
- Flash (code)	-1.0%	-93.73%
- RAM (données)	-0.65%	-98.1%

la partition racine, l'empreinte mémoire de Pip-MPU comprend les métadonnées utiles pour la création d'autres partitions, dont la partition enfant de notre scénario. Les métadonnées des partitions comprennent les structures encapsulant la liste des blocs mémoire et leurs attributs, ainsi que des données globales à la partition. Lorsque le nombre de blocs d'une partition augmente, ces blocs sont référencés dans des structures de taille S. Chaque structure peut référencer un nombre constant de blocs C. Ainsi, pour une partition de B blocs, nous obtenons l'empreinte mémoire de ses métadonnées par la formule

$$K + (B \bmod C) \times S \text{ octets}$$

avec K une taille de métadonnées incompressible. Dans notre implémentation, $K = 640$, $C = 8$, $S = 512$. Puisque chaque partition a besoin d'une structure de métadonnées au minimum pour référencer les premiers blocs, l'empreinte mémoire d'une partition en RAM sera d'au minimum 1152 octets (liste initiale de $B = 8$ blocs). Par ailleurs, le nombre de structures de métadonnées pour une partition donnée est limité à $MaxS$ lors de la compilation, ce qui implique une empreinte mémoire maximum de $K + MaxS \times S$ pour cette partition. Pour notre système, $MaxS = 8$ par partition, soit $640 + 8 \times 512 = 4736$ octets. De plus, $MaxS$ dicte le nombre de blocs qu'une partition peut référencer au maximum par $C \times MaxS$, soit ici 64 blocs.

Pour obtenir les métriques de performances du Tableau 2 (étendu dans les annexes), nous exécutons l'application pour chaque scénario (référence, partition racine et partition enfant). Nous lançons le programme principal 3 fois consécutivement pour la même expérience, ce qui nous permet de récolter des données sur au moins 20 secondes (chaque application s'éxécutant en 5-7 secondes). Nous lançons l'expérience 5 fois et nous effectuons ensuite des statistiques (moyenne μ et écart-type σ) que nous moyennons pour chaque application. Le surcoût indiqué est la moyenne du surcoût observé pour chaque scénario comparé au scénario de référence.

Le nombre de cycles est obtenu par l'unité du processeur *Data Watchpoint and Trace* (DWT). Nous initialisons le compteur juste avant le lancement de l'application et nous récupérons le nombre de cycles écoulés après la fin de la phase d'initialisation ainsi que lorque l'application a terminé son travail. La fin de la phase d'initialisation marque le début de la phase de test qui lance l'application. De plus, le scénario de référence s'exécute toujours en mode privilégié,

ce qui correspond à 100% de cycles privilégiés. A l'inverse, dans les scénarios Pip, les cycles privilégiés sont tous ceux mesurés dans les routines de Pip-MPU. Nous indiquons le ratio du nombre de cycles privilégiés sur le nombre total de cycles 1) depuis le démarrage de la partition racine 2) seulement durant l'exécution de l'application.
Les zones de mémoire accessible représentent la mémoire que la partition peut accéder. L'application dans le scénario de référence a accès à toute la mémoire, soit 100%, tandis que dans les scénarios Pip les zones accessibles sont restreintes aux blocs de l'espace mémoire. Pour la partition racine, la mémoire accessible correspond à toute la mémoire, en retirant Pip-MPU et les composants de démarrage. A partir de là, la partition racine, ainsi que n'importe quelle autre partition parent, décide quels blocs mémoire sont partagés avec les partitions enfant, contrôlant de cette manière leurs zones de mémoire accessible.
La consommation énergétique est mesurée en utilisant un Power Profiler Kit I (PPKI) [7] monté sur la puce nRF52840-DK. Le PPK produit des mesures de courant à 77 kHz en moyennant sur 4 mesures, que nous multiplions par une tension fixe et que nous intégrons sur le temps écoulé afin d'obtenir la consommation énergétique totale. Etant donné l'utilisation du *semihosting* pour renvoyer les données de performance à l'ordinateur pour analyse, le débuggeur reste actif. Néanmoins, aucune entrée/sortie n'a lieu pendant la phase de test.

3.3. Discussions et limitations de l'approche

L'évaluation produit des informations précieuses pour considérer un portage sur Pip-MPU.
Pip-MPU prend respectivement 1664 octets (données, pile, métadonnées de la partition racine) et 9544 octets (code) des 256 ko de RAM et 1 Mo de Flash disponible. Cela respecte aisément les contraintes de nos cibles (environ 3.3% de la RAM et 3.8% de la Flash des appareils de Classe 2 IETF) et laisse assez de place pour des applications plus complexes. Nous espérons ainsi un bon ratio entre la taille de Pip-MPU relativement à la taille d'un système d'exploitation porté sur Pip-MPU avec ses applications. Toutefois, nous avons fait en sorte de ne pas déclencher de faute mémoire lors de l'exécution de l'application. Cela aurait eu comme conséquence d'augmenter la taille de la pile à cause de la routine de reconfiguration partielle (cf *framework*).
Les zones de mémoire accessible montrent l'étendue de la surface d'attaque. En utilisant Pip-MPU, l'application est non privilégiée et est limitée par la MPU, contrairement au scénario de référence où l'application est entièrement privilégiée. Pour la partition racine, la mémoire accessible diminue de 1%. Cependant, plus la partition est éloignée de la partition racine, plus son espace mémoire se réduit et meilleure devient la métrique. Pour la partition enfant de notre implémentation, nous avons réduit sa mémoire accessible de respectivement 93% et 98% sur les mémoire Flash et RAM, soit autant de réduction par rapport au scénario de référence.
L'évaluation révèle une empreinte mémoire de 1 Ko dans une partition parent pour chaque nouvelle partition enfant créée. Ce minimum doit en fait être relevé par le nombre d'entrées utilisées dans la partition racine pour référencer les métadonnées de la partition enfant. Les entrées additionnelles ne trouvent pas forcément de places disponibles dans les structures de métadonnées qui référencent les attributs des blocs, ce qui peut causer la création de nouvelles structures de métadonnées dans le parent pour contenir ces entrées (512 octets supplémentaires par nouvelle structure dans notre implémentation).
Par ailleurs, nous observons un surcoût de cycles lors de la phase de test d'environ 16% causé par la restoration de contexte de Pip-MPU lorsque l'interruption SysTick est déclenchée. Cette valeur doit être appréciée dans le cadre de l'évaluation et ces valeurs sont attendues plus élevées pour un portage de système d'exploitation sur Pip-MPU à cause de multiples causes d'interruptions. Ces performances sont suffisantes pour notre évaluation, mais il existe de la latitude pour des améliorations en optimisant les routines d'interruptions, par exemple en ajoutant

des structures de données d'optimisation similaires à ce qui existe dans Pip (version MMU). De plus, Pip force l'application à s'exécuter dans l'espace utilisateur (100% non privilégié). Nous observons bien une chute de plus de 99% du nombre de cycles privilégiés, le pourcentage restant correspondant à Pip. Par ailleurs, le nombre de régions MPU (8) n'a pas de conséquence dans notre évaluation puisqu'une seule structure de métadonnées suffit pour porter les blocs de l'espace mémoire. S'il y avait eu plusieurs partitions enfant, une seule structure n'aurait pas suffit, ce qui aurait comme conséquence la création de structures supplémentaires.

La consommation d'énergie a augmenté de 16-18% en utilisant Pip-MPU. Par ailleurs, nous avons lancé les tests avec Pip-MPU mais en désactivant la MPU. Cela a diminué la consommation énergétique de 0.02-0.2%, correspondant à la consommation seule de la MPU. Cela indique que l'utilisation de la MPU (lors de changement de contexte et protection permanente) n'impacte pas significativement la consommation énergétique globale. Ces mesures sont importantes pour les objets connectés seulement alimentés par batterie.

D'autres métriques sont proposées dans BenchIoT mais ne sont pas reportées ici pour les raisons suivantes.

Premièrement, nous n'avons pas évalué le nombre de cycles de veille puisque Pip-MPU ne met jamais le CPU dans ce mode.

Deuxièmement, nous n'avons pas inclus la métrique *Data Execution Prevention* (DEP) ou la mise en vigueur du principe *W^X* puisque Pip ne les met pas en place. En effet, l'existence de ces principes de sécurité ou d'autres (comme décider quels blocs mémoire il faut isoler) sont des choix de conception revenant aux partitions elles-mêmes.

Troisièmement, les gadgets ROP et appels indirects sont des techniques d'attaques connues pour prendre le contrôle du flot de contrôle [28]. Cependant, nous ne les considérons pas pertinentes pour Pip-MPU. La raison est que les tentatives d'accès illégitimes causées par l'utilisation de gadgets ROP finissent en des fautes mémoire MPU gérées directement par les ancêtres. Il est important de noter que Pip n'empêche pas les attaques ROP à l'intérieur d'une partition mais contre Pip et les partitions ancêtres. Par ailleurs, les services principaux de Pip étant développés en Coq avant sa translation en C, il porte les caractéristiques d'un language de programmation fonctionnel tel qu'une utilisation élevée de la pile et beaucoup de fonctions, ce qui dégrade ces métriques.

Quatrièmement, nous n'avons pas distingué les cycles privilégiés des cycles SVC puisque Pip est seul privilégié. En effet, les points d'entrée de Pip sont toutes les interruptions, dont les SVC, qui forment l'unique code privilégié qui peut s'exécuter après la phase d'initialisation.

4. Conclusion

Dans ce papier, nous avons évalué Pip-MPU, le variant de Pip basé sur la MPU qui ne requiert aucune modification matérielle sur des produits informatiques COTS (*commercial off-the-shelf*), ce qui maintient les coûts de production bas. L'évaluation de notre prototype implémenté pour un SOC intégrant un processeur ARMv7 Cortex-M montre que Pip-MPU réduit la mémoire accessible d'une application de 100% à 2% en nécessitant 10 Ko de Flash, 550 octets de RAM et impliquant un surcoût de 16% sur les cycles et sur la consommation énergétique. Nous avons montré que des simples applications comme celles utilisées dans notre évaluation peuvent directement bénéficier de la protection de Pip-MPU moyennant peu d'efforts.

Pip-MPU est actuellement en cours de vérification formelle en suivant l'approche développée pour Pip. Les travaux (base de code, évaluation, vérification formelle) sont en cours de publication en open-source. Nous explorerons dans de futurs travaux comment la flexibilité de Pip-MPU peut être utilisée pour créer des architectures sécurisées dès la conception sur des conteneurs pour objets contraints, comme décrits dans [11].

Remerciements

Les travaux de recherche menant à ces résultats ont été partiellement finances par le programme bilatéral franco-allemand MESRI-BMBF sur la cybersécurité sous les conventions n° ANR-20-CYAL-0005 et 16KIS1395K. L'article reflète uniquement les vues des auteurs. MESRI et BMBF ne sont pas responsables des utilisations qui peuvent être faites à partir des informations qu'il contient. Ces travaux ont également été partiellement financés par IRCICA, USR-3380 (Lille, France).

Bibliographie

1. ARM . – Website of : Trustzone for cortex-m (arm). – https://www.arm.com/technologies/trustzone-for-cortex-m/, 2022. [Online ; accessed March21, 2022].
2. Free and Open Source Silicon Foundation . – Website of : Embench™. – https://github.com/embench/embench-iot/, 2022. [Online ; accessed March 18, 2022].
3. IETF . – Website of : Terminology for constrained-node networks (ietf). – https://datatracker.ietf.org/doc/html/rfc7228\#page-8, 2022. [Online ; accessed March 22, 2022].
4. Intel . – Website of : Intel® software guard extensions (intel® sgx). – https://www.intel.co.uk/content/www/uk/en/architecture-and-technology/software-guard-extensions.html/, 2021. [Online ; accessed June 14, 2021].
5. Nordic Semiconductor . – Website of : nrf52840 dk (nordic semiconductor). – https://www.nordicsemi.com/Products/Development-hardware/nrf52840-dk/, 2022. [Online ; accessed March 18, 2022].
6. Nordic Semiconductor . – Website of : Power profiler kit ii (ppkii). – https://www.nordicsemi.com/Products/Development-hardware/Power-Profiler-Kit-2/, 2022. [Online ; accessed April 13, 2022].
7. Nordic Semiconductor . – Website of : Power profiler kit (ppk). – https://www.nordicsemi.com/Products/Development-hardware/Power-Profiler-Kit/, 2022. [Online ; accessed March 18, 2022].
8. Wilgaard . – Website of : ppk_api (github). – https://github.com/wlgrd/ppk_api, 2022. [Online ; accessed March 23, 2022].
9. Almakhdhub (N. S.), Clements (A. A.), Payer (M.) et Bagchi (S.). – BenchIoT : A Security Benchmark for the Internet of Things. *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2019, pp. 234–246.
10. Ammar (M.), Crispo (B.), Jacobs (B.), Hughes (D.) et Daniels (W.). – SV—The Security MicroVisor : A Formally-Verified Software-Based Security Architecture for the Internet of Things. *IEEE Transactions on Dependable and Secure Computing*, vol. 16, n5, 2019, pp. 885–901.
11. Baccelli (E.), Doerr (J.), Kikuchi (S.), Padilla (F. A.), Schleiser (K.) et Thomas (I.). – Scripting Over-The-Air : Towards Containers on Low-end Devices in the Internet of Things. *2018 IEEE International Conference on Pervasive Computing and Communications Workshops, PerCom Workshops 2018*, 2018, pp. 504–507.
12. 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.
13. Brasser (F. N.), El Mahjoub (B.), Sadeghi (A. R.), Wachsmann (C.) et Koeberl (P.). – TyTAN : Tiny trust anchor for tiny devices. *Proceedings - Design Automation Conference*, vol. 2015-July, 2015.

14. 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.
15. Clements (A. A.), Almakhdhub (N. S.), Saab (K. S.), Srivastava (P.), Koo (J.), Bagchi (S.) et Payer (M.). – Protecting Bare-Metal Embedded Systems with Privilege Overlays. *Proceedings - IEEE Symposium on Security and Privacy*, 2017, pp. 289–303.
16. Dejon (N.), Gaber (C.) et Grimaud (G.). – Compartimentation dynamique imbriquée pour objets contraints. – In *Conférence francophone d'informatique en Parallélisme, Architecture et Système (COMPAS 2021)*, Lyon (en virtuel), France, juillet 2021.
17. Dejon (N.), Gaber (C.) et Grimaud (G.). – Nested compartmentalisation for constrained devices. *Proceedings - 2021 International Conference on Future Internet of Things and Cloud, FiCloud 2021*, 2021, pp. 334–341.
18. Grisafi (M.), Ammar (M.), Roveri (M.) et Crispo (B.). – PISTIS : Trusted Computing Architecture for Low-end Embedded Systems. *USENIX - USENIX Security Symposium*, 2022.
19. Gu (R.), Shao (Z.), Chen (H.), Wu (X.), Kim (J.), Sjöberg (V.) et Costanzo (D.). – CertiKOS : An extensible architecture for building certified concurrent OS kernels. *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016*, vol. 4, 2016, pp. 653–669.
20. Jomaa (N.), Torrini (P.), Nowak (D.), Grimaud (G.) et Hym (S.). – Proof-Oriented Design of a Separation Kernel with Minimal Trusted Computing Base. vol. 076, 2018.
21. 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.
22. Klein (G.), Elphinstone (K.), Heiser (G.), Andronick (J.), Cock (D.), Derrin (P.), Elkaduwe (D.), Engelhardt (K.), Kolanski (R.), Norrish (M.), Sewell (T.), Tuch (H.) et Winwood (S.). – SeL4 : Formal verification of an OS kernel. *SOSP'09 - Proceedings of the 22nd ACM SIGOPS Symposium on Operating Systems Principles*, 2009, pp. 207–220.
23. 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.
24. Levy (A.), Giffin (D. B.), Campbell (B.), Pannuto (P.), Levis (P.), Ghena (B.) et Dutta (P.). – Multiprogramming a 64 kB Computer Safely and Efficiently. *SOSP 2017 - Proceedings of the 26th ACM Symposium on Operating Systems Principles*, 2017, pp. 234–251.
25. Noorman (J.), Agten (P.), Daniels (W.), Strackx (R.), Van Herrewege (A.), Huygens (C.), Preneel (B.), Verbauwhede (I.), Piessens (F.), Herrewege (A. V.), Huygens (C.), Preneel (B.), Verbauwhede (I.) et Piessens (F.). – Sancus : Low-cost Trustworthy Extensible Networked Devices with a Zero-software Trusted Computing Base. *Proceedings of the USENIX Security Symposium*, 2013, pp. 479–494.
26. 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.
27. Perito (D.), Tsudik (G.), Defrawy (K. E.) et Francillon (A.). – SMART : Secure and Minimal Architecture for (Establishing a Dynamic) Root of Trust. *Ndss*, vol. 12, 2015, pp. 1–15.
28. Roemer (R.), Buchanan (E.), Shacham (H.) et Savage (S.). – Return-oriented programming : Systems, languages, and applications. *ACM Transactions on Information and System Security*, vol. 15, n1, 2012, pp. 1–34.
29. Xia (H.), Woodruff (J.), Barral (H.), Esswood (L.), Joannou (A.), Kovacsics (R.), Chisnall (D.), Roe (M.), Davis (B.), Napierala (E.), Baldwin (J.), Gudka (K.), Neumann (P. G.), Richardson (A.), Moore (S. W.) et Watson (R. N.). – CheriRTOS : A Capability Model for Embedded

Devices. *Proceedings - 2018 IEEE 36th International Conference on Computer Design, ICCD 2018*, 2019, pp. 92–99.

30. Zhou (X.), Li (J.), Zhang (W.), Zhou (Y.), Shen (W.) et Ren (K.). – Opec : Operation-based Security Isolation for Bare-metal Embedded Systems. *Seventeenth European Conference on Computer Systems (EuroSys '22), April 5-8, 2022, RENNES, France*, vol. 1, n1, 2022, pp. 317–333.

Annexes

A. Evaluation

TABLE 3 – Comparaison des performances et de la sécurité (par rapport à la référence). Tableau étendu du Tableau 2.

Métriques	**Partition racine**	**Partition enfant**
Cycles		
Cycles :		
i) au total	$\mu = 76302131$	$\mu = 74538344$
	$\sigma = 67494444$	$\sigma = 73634323$
	(+16.31%)	(+16.4%)
ii) pendant le test	$\mu = 76203107$	$\mu = 74372762$
	$\sigma = 67495112$	$\sigma = 73634647$
	(+16.29%)	(+16.36%)
Ratio des cycles privilégiés sur le total des cycles :		
i) au total	$\mu = 0.86\%$	$\mu = 0.92\%$
	$\sigma = 3.8 \times 10^{-5}\%$	$\sigma = 3.3 \times 10^{-5}\%$
	(-99.14%)	(-99.08%)
ii) pendant le test	$\mu = 0.87\%$	$\mu = 0.92\%$
	$\sigma = 3.9 \times 10^{-5}\%$	$\sigma = 3.3 \times 10^{-5}\%$
	(-99.13%)	(-99.08%)
Consommation énergétique pendant la phase de test		
Totale	$\mu = 24.76$ mJ	$\mu = 26.6$ mJ
	$\sigma = 22.42$ mJ	$\sigma = 23.00$ mJ
	$(+16.7\%)$	$(+18.4\%)$
dont causée par la MPU	$\mu = 0.05$ mJ	$\mu = 0.07$ mJ
	$\sigma = 0.16$ mJ	$\sigma = 0.11$ mJ
	$(+0.03\%)$	$(+0.04\%)$
Securité		
Ratio de la mémoire accessible depuis l'application sur la mémoire totale :		
- Flash (code)	99.0%	6.27%
	(-1.0%)	(-93.73%)
- RAM (données)	99.35%	1.9%
	(-0.65%)	(-98.1%)

B. Détails techniques pour la mise en place du Power Profiler Kit (PPK)

Notre banc d'essai illustré en Figure 1 comporte une puce nRF52840-DK sur laquelle s'exécute Pip-MPU et l'application de test, ainsi qu'une puce nRF52840-DK additionnelle qui s'interface avec le PPK pour le piloter et récupérer ses données. Nous avons utilisé une librairie pour le PPK [8] afin de déclencher les mesures car l'application de bureau n'était pas assez stable pour nos expériences et cela facilitait par ailleurs l'intégration avec nos scripts Python. Toutefois, une nouvelle version du PPK (PPKII [6]) existe depuis quelques années, ce qui a causé l'arrêt de la maintenance de la librairie et l'intégration à nos scripts a nécessité de retrouver la bonne combinaison entre le micro-code du PPK et la librairie ainsi qu'avec ses dépendances.

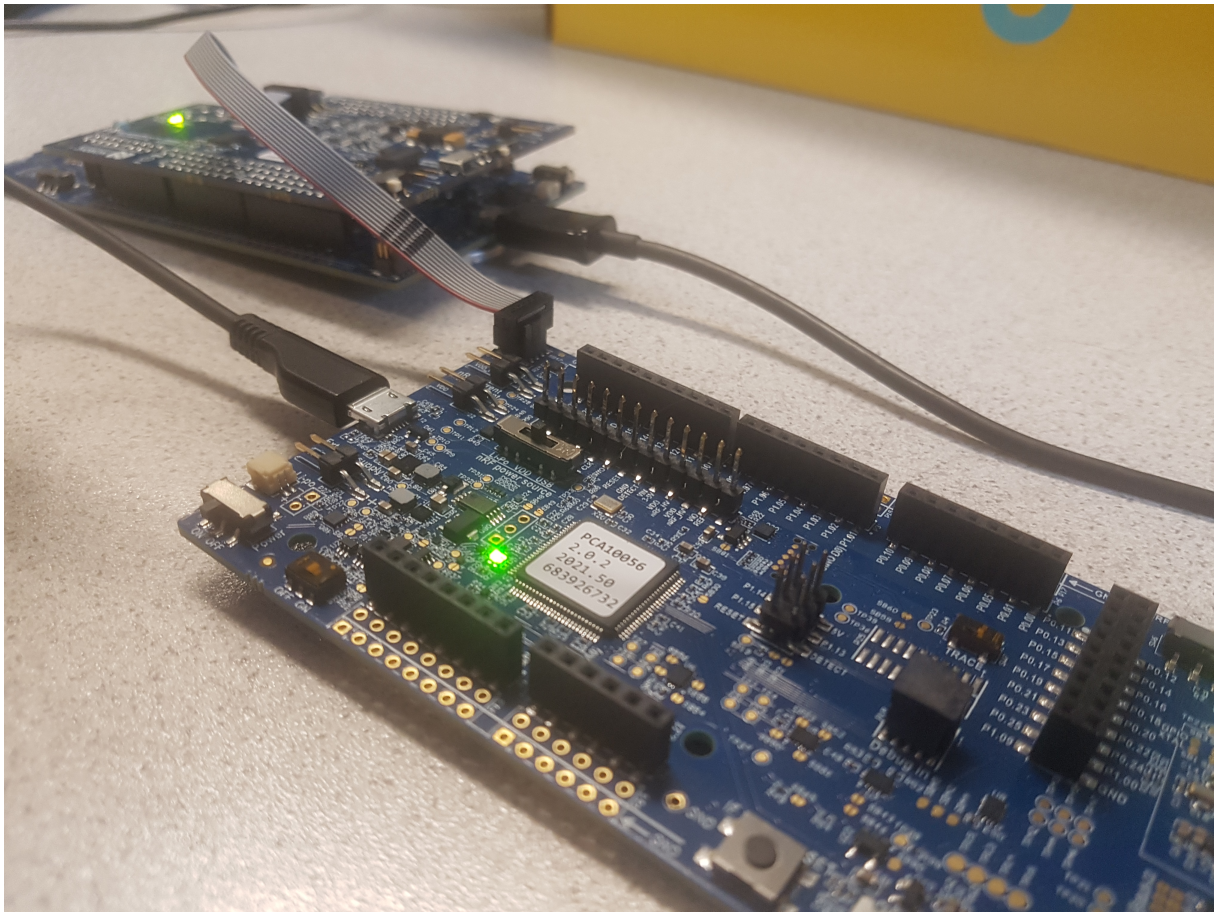

FIGURE 1 – Banc d'essai. Au premier plan, la nRF52840-DK qui contrôle le PPK. Au second plan, la nRF52840-DK où s'exécute l'application de test et sur laquelle est montée le PPK.

C. Consommation énergétique en fonction des scénarios

Pour notre analyse, nous ne gardons que la consommation énergétique durant la phase de test. Pour cela, nous marquons cette phase en plaçant le processeur en mode veille avant et après la phase de test en sortant le processeur de ce mode par une minuterie externe. De cette manière, nous pouvons facilement identifier la phase de test par la mesure du courant qui chute énormément en mode veille (environ 6 mA pendant la phase de test jusqu'à quelques µA en mode veille) comme nous pouvons l'apercevoir sur les figures 2, 3 et 4. La partie rouge des figures est le tracé durant la phase de test. Sur ces figures, nous distinguons nettement les chutes de consommation avant et après cette phase. Les figures représentent les variations de consommation énergétique durant une expérience en fonction des différents scénarios en considérant la même application.

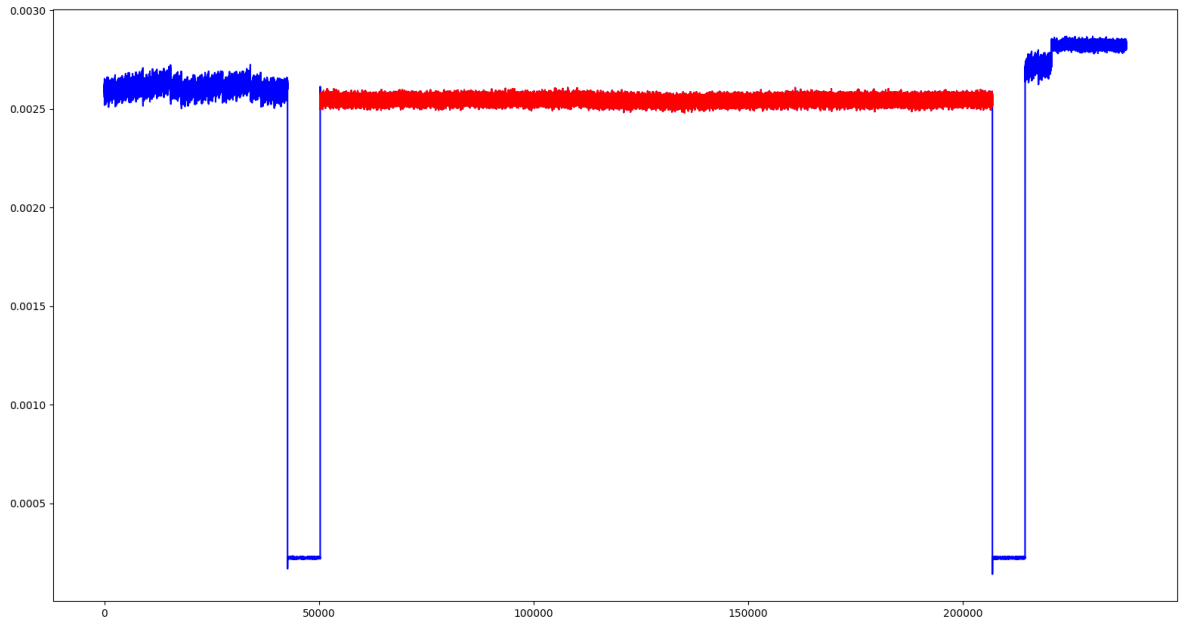

FIGURE 2 – Scénario de référence. Consommation énergétique (mJ) en fonction du temps (échantillons).

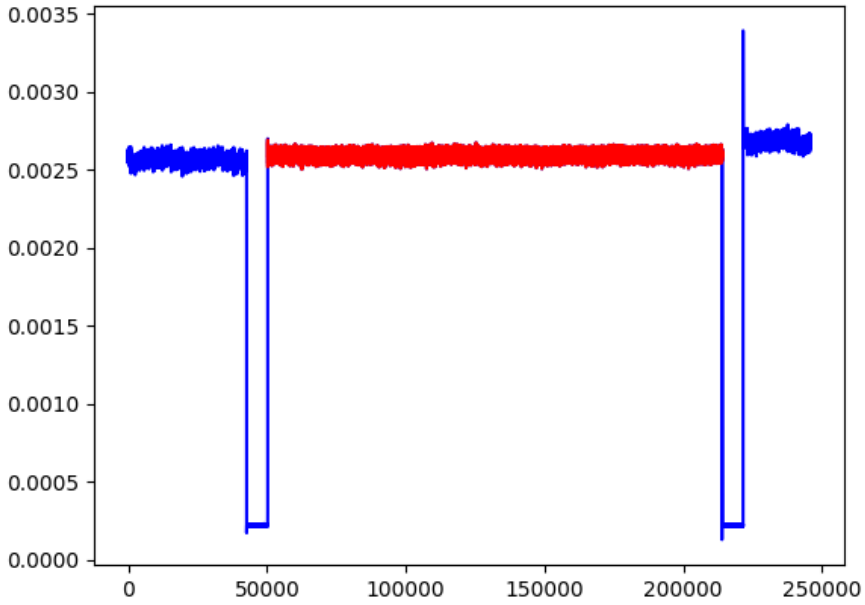

FIGURE 3 – Scénario Pip partition racine. Consommation énergétique (mJ) en fonction du temps (échantillons).

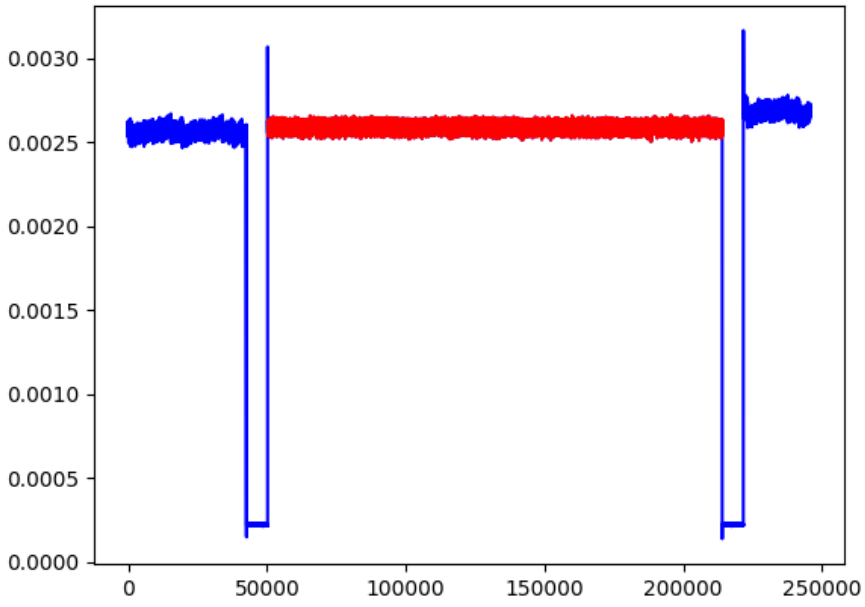

FIGURE 4 – Scénario Pip partition enfant. Consommation énergétique (mJ) en fonction du temps (échantillons).