



# **Durcissement d'une implémentation de la machine virtuelle Java Card grâce à l'UPM**

Guillaume BOUFFARD ([guillaume.bouffard@ssi.gouv.fr](mailto:guillaume.bouffard@ssi.gouv.fr))   Léo GASPARD ([leo@gaspard.io](mailto:leo@gaspard.io))

Agence nationale de la sécurité des systèmes d'information

SSTIC 2018

# La Racine de confiance

---

- **Plusieurs fonctionnalités** ont besoin d'un environnement de confiance où il est possible :
  - ▶ de **stocker** des données sensibles :
    - en protégeant la confidentialité/intégrité des données;
  - ▶ d'**exécuter** des opérations sensibles :
    - sans aucune **fuite**.
- La **racine de confiance** est un environnement d'exécution sécurisé.

# La Racine de confiance

---

- **Plusieurs fonctionnalités** ont besoin d'un environnement de confiance où il est possible :
  - ▶ de **stocker** des données sensibles :
    - en protégeant la confidentialité/intégrité des données;
  - ▶ d'**exécuter** des opérations sensibles :
    - sans aucune **fuite**.
- La **racine de confiance** est un environnement d'exécution sécurisé.
- Traditionnellement, c'est un **composant sécurisé**.

# La Racine de confiance

---

- **Plusieurs fonctionnalités** ont besoin d'un environnement de confiance où il est possible :
  - ▶ de **stocker** des données sensibles :
    - en protégeant la confidentialité/intégrité des données;
  - ▶ d'**exécuter** des opérations sensibles :
    - sans aucune **fuite**.
- La **racine de confiance** est un environnement d'exécution sécurisé.
- Traditionnellement, c'est un **composant sécurisé**. Mais ça peut aussi être :
  - ▶ l'émulation de composant matériel sécurisé, (type **enclaves sécurisées**)
  - ▶ de la **cryptographie en boîte blanche**.

# Le Logiciel dans les composants sécurisés

---

- 1 Le développeur a un accès direct aux fonctionnalités bas-niveau.
- 2 Le développeur utilise une interface liant les fonctionnalités matérielles et logicielles.

# Le Logiciel dans les composants sécurisés

---

- 1 Le développeur a un accès direct aux fonctionnalités bas-niveau.
  - ▶ Interface de Programmation Applicative (IPA) propriétaire.
  - ▶ Dépendant du matériel et du fabricant.
  - ▶ Développement bas-niveau.
  - ▶ **Accord de confidentialité** requis pour accéder aux fonctionnalités bas-niveaux d'un composant certifié.
- 2 Le développeur utilise une interface liant les fonctionnalités matérielles et logicielles.

# Le Logiciel dans les composants sécurisés

---

- 1 Le développeur a un accès direct aux fonctionnalités bas-niveau.
  - ▶ Interface de Programmation Applicative (IPA) propriétaire.
  - ▶ Dépendant du matériel et du fabricant.
  - ▶ Développement bas-niveau.
  - ▶ Accord de confidentialité requis pour accéder aux fonctionnalités bas-niveaux d'un composant certifié.
- 2 Le développeur utilise une interface liant les fonctionnalités matérielles et logicielles.
  - ▶ IPA standardisée.
  - ▶ Indépendance entre la couche matérielle et logicielle.
  - ▶ Aucune connaissance nécessaire sur la couche matérielle,
    - Aucun accord de confidentialité n'est donc requis.

# Le Logiciel dans les composants sécurisés

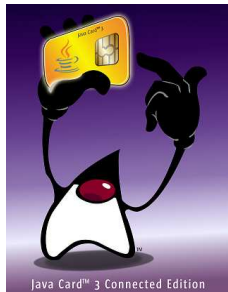
---

- 1 Le développeur a un accès direct aux fonctionnalités bas-niveau.
  - ▶ Interface de Programmation Applicative (IPA) propriétaire.
  - ▶ Dépendant du matériel et du fabricant.
  - ▶ Développement bas-niveau.
  - ▶ Accord de confidentialité requis pour accéder aux fonctionnalités bas-niveaux d'un composant certifié.
- 2 Le développeur utilise une interface liant les fonctionnalités matérielles et logicielles.
  - ▶ IPA standardisée.
  - ▶ Indépendance entre la couche matérielle et logicielle.
  - ▶ Aucune connaissance nécessaire sur la couche matérielle,
    - Aucun accord de confidentialité n'est donc requis.
  - ▶ De nombreuses technologies disponibles : Windows pour carte à puce, MULTOS, **Java Card**, ...



# La Technologie Java Card

- Basée sur une machine virtuelle :
  - ▶ applications développées en Java,
  - ▶ multi-applicative (*mais pas multitâche*),
  - ▶ se veut *aussi sécurisée que la machine virtuelle Java*.
- Spécification & chaîne de compilation disponibles gratuitement.
- Pas d'implémentation complète à code source ouvert.



# La Technologie Java Card

- Basée sur une machine virtuelle :
  - ▶ applications développées en Java,
  - ▶ multi-applicative (mais pas multitâche),
  - ▶ se veut *aussi sécurisée que la machine virtuelle Java*.
- Spécification & chaîne de compilation disponibles gratuitement.
- Pas d'implémentation complète à code source ouvert.
- La majorité des plate-formes Java Card sont évaluées :
  - ▶ Environ **200 rapports d'évaluations** dans le schéma Critères Communs ont eu lieu en EUROPE en 2017 (dans le groupement SOG-IS).



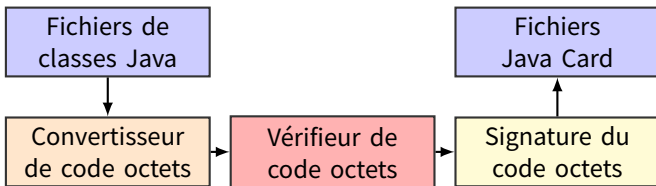
# La Technologie Java Card

- Basée sur une machine virtuelle :
  - ▶ applications développées en Java,
  - ▶ multi-applicative (mais pas multitâche),
  - ▶ se veut *aussi sécurisée que la machine virtuelle Java*.
- Spécification & chaîne de compilation disponibles gratuitement.
- Pas d'implémentation complète à code source ouvert.
- La majorité des plate-formes Java Card sont évaluées :
  - ▶ Environ **200 rapports d'évaluations** dans le schéma Critères Communs ont eu lieu en EUROPE en 2017 (dans le groupement SOG-IS).
  - ▶ **Dont environ 130 évaluations réalisées en FRANCE.**

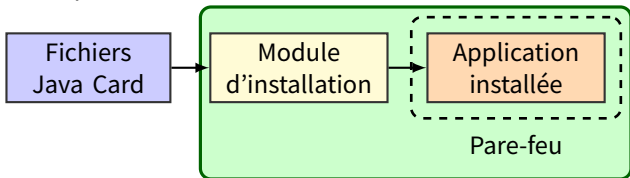


# Le Modèle de sécurité Java Card

## ■ La chaîne de compilation :



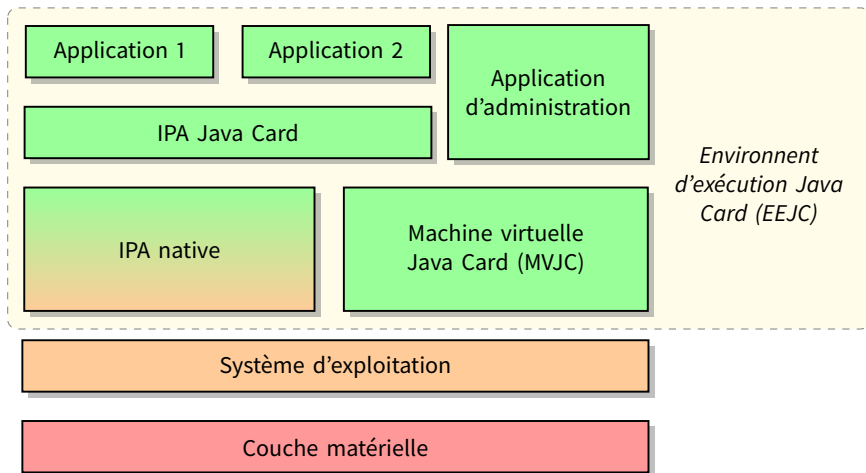
## ■ Sécurité embarquée dans la machine virtuelle :



Machine virtuelle Java Card

*(évaluée dans le cadre  
des Critères Communs)*

# La Machine virtuelle Java Card



# Motivations

---

## Les implémentations existantes sont génériques

- Confinement des applications Java Card grâce à la MVJC.
- Indépendance entre les fonctionnalités matérielles et la MVJC.

# Motivations

---

## Les implémentations existantes sont génériques

- Confinement des applications Java Card grâce à la MVJC.
- Indépendance entre les fonctionnalités matérielles et la MVJC.

## Objectif de notre travail

Confiner les applications Java Card grâce aux **mécanismes matériels**.

# Motivations

---

## Les implémentations existantes sont génériques

- Confinement des applications Java Card grâce à la MVJC.
- Indépendance entre les fonctionnalités matérielles et la MVJC.

## Objectif de notre travail

Confiner les applications Java Card grâce aux **mécanismes matériels**.

## Conséquence

Casser la généralité pour améliorer la sécurité de notre implémentation.



# Les Mécanismes de sécurité matériels sur les composants

---

Les CPU récents embarquent des mécanismes physiques de protection mémoire :

- Unité de Gestion Mémoire
  - ▶ Sur les composants à forte contrainte en ressources : Unité de Protection Mémoire (UPM)
- Unité de Gestion Mémoire pour les Entrées-Sorties

## Unité de Protection Mémoire

---

- Modifiable seulement en mode privilégié (heureusement!)
- Permet de fixer des contraintes de lecture, écriture et exécution

## Unité de Protection Mémoire

---

- Modifiable seulement en mode privilégié (heureusement!)
- Permet de fixer des contraintes de lecture, écriture et exécution
- 8 régions de taille  $2^n$  alignées naturellement
- (chacune pouvant être divisée en 8 sous-régions de même longueur)

## Fonctionnement de l'UPM

---

**Données :** Une opération accédant à une *adresse* avec une *permission* (lecture, écriture ou exécution)

**Résultat :** Autorisation ou rejet

résultat  $\leftarrow$  Refuser;

**si** CarteParDéfautActivée **alors**

| résultat  $\leftarrow$  AutoriseAccès(*permission*, EstPrivilégié, CarteParDéfaut)

**fin**

**pour**  $i \leftarrow 0$  à 7 **faire**

| **si** *adresse*  $\in$  Région( $i$ ) **alors**

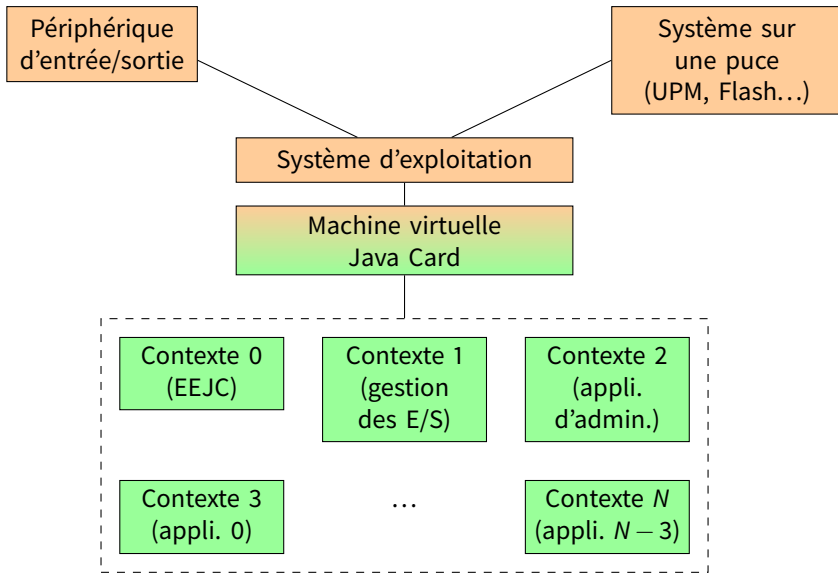
| | résultat  $\leftarrow$  AutoriseAccès(*permission*, EstPrivilégié, Région( $i$ ))

| **fin**

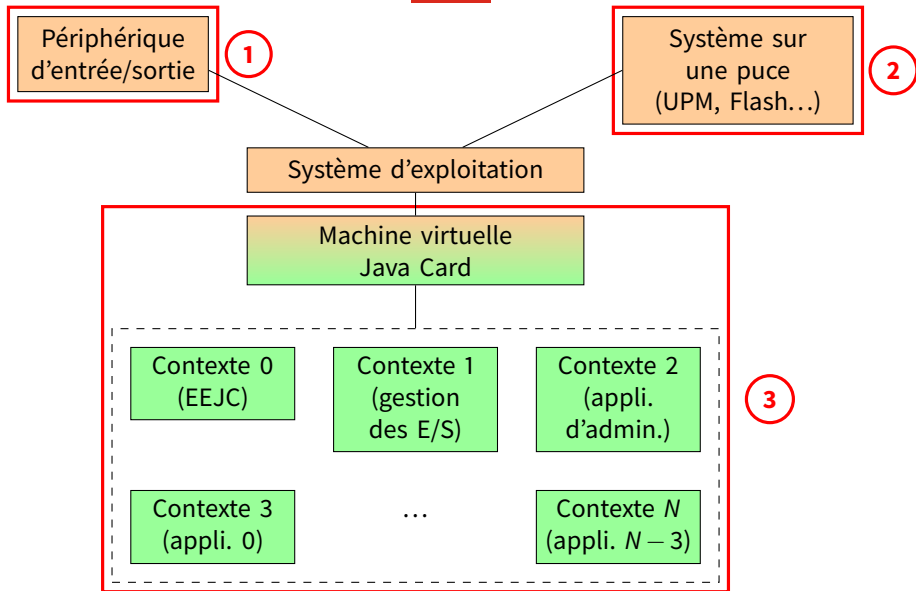
**fin**

**retourner** résultat

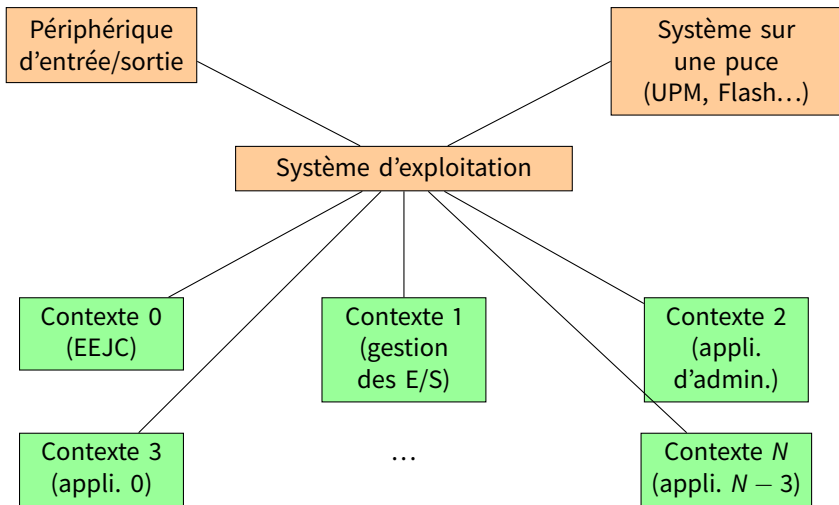
# Implémentation classique du modèle d'une MVJC



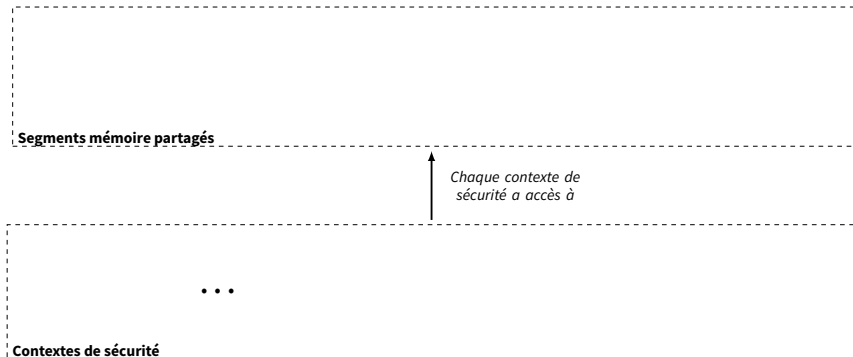
# Implémentation classique du modèle d'une MVJC



# Architecture de notre implémentation de la MVJC

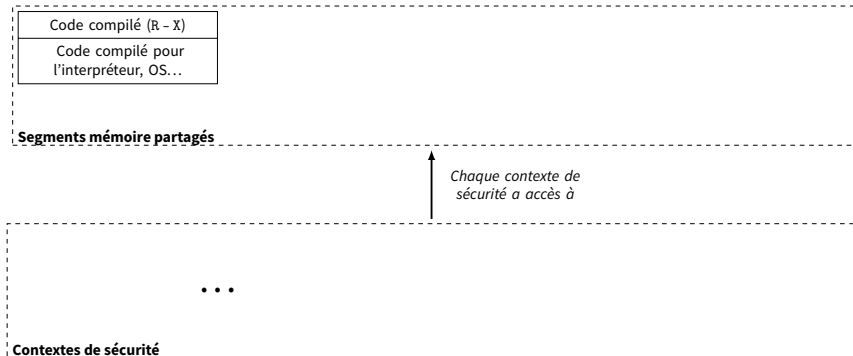


# Utilisation de l'UPM dans la MVJC

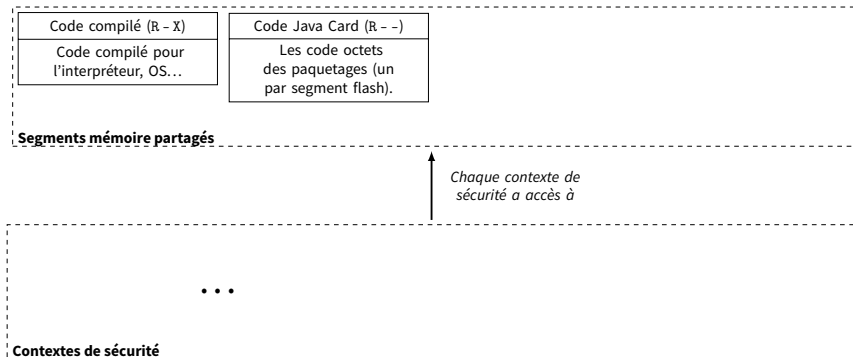




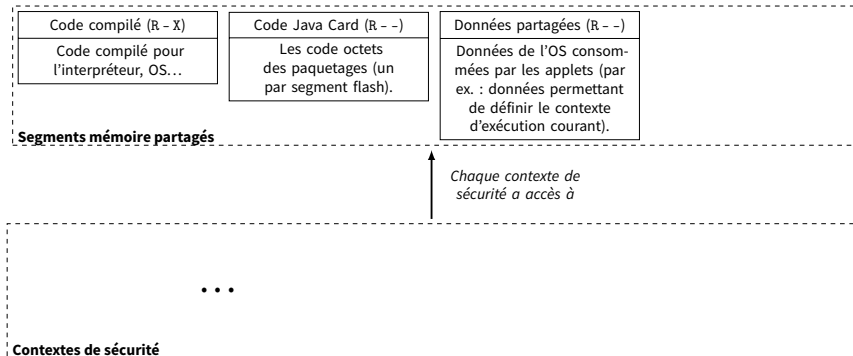
# Utilisation de l'UPM dans la MVJC



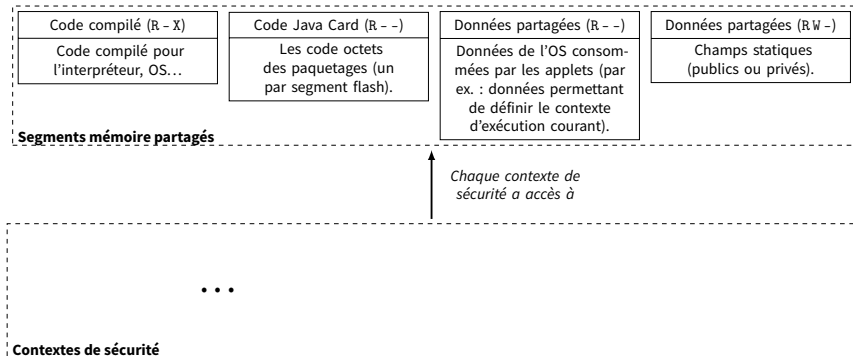
# Utilisation de l'UPM dans la MVJC



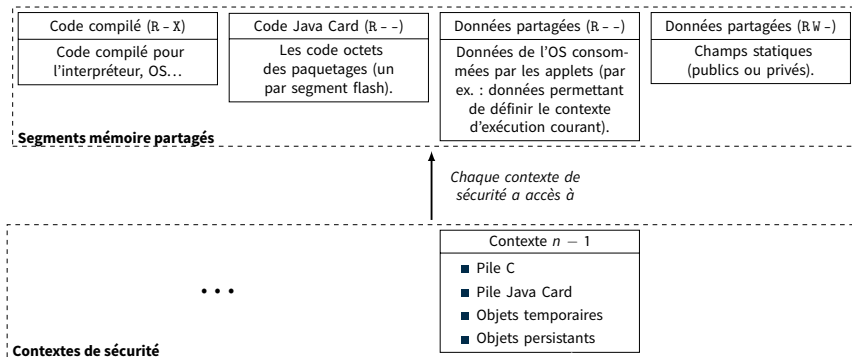
# Utilisation de l'UPM dans la MVJC



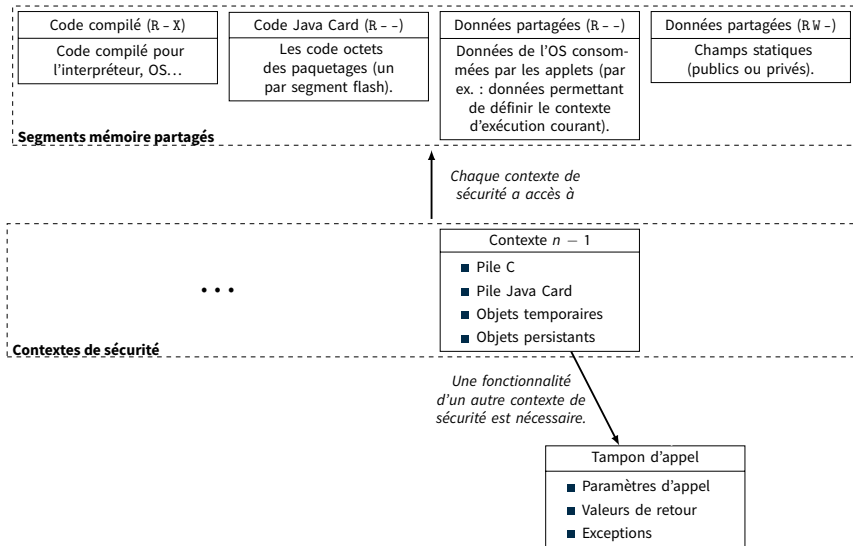
# Utilisation de l'UPM dans la MVJC



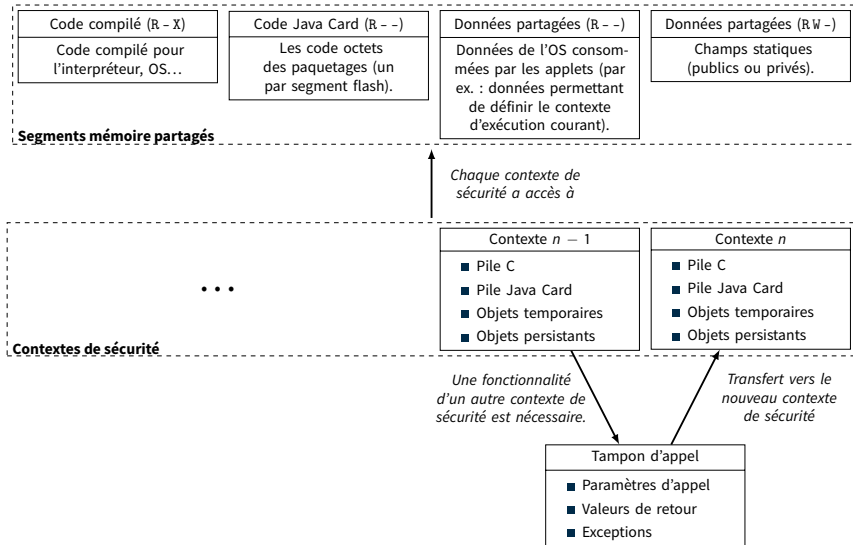
# Utilisation de l'UPM dans la MVJC



# Utilisation de l'UPM dans la MVJC



# Utilisation de l'UPM dans la MVJC



## Plate-forme cible

---

### STM32F401RE

- 512 Kio de mémoire flash
- 96 Kio de RAM
- ARM Cortex-M4 (84 MHz)



## Plate-forme cible

---

### STM32F401RE

- 512 Kio de mémoire flash
- 96 Kio de RAM
- ARM Cortex-M4 (84 MHz)

### Mécanismes de sécurité embarqués

- Ségrégation des privilèges entre modes privilégié et non-privilégié
- **Une Unité de Protection Mémoire (UPM)**

## Rust?

---

- gestion mémoire sans ramasse-miette,
- typage fort,
- sûreté des fils d'exécution,
- gestion des erreurs,
- code natif avec possibilité de zéro-copie,
- intégration facile avec du code C et C++,
- isolation des fragments `unsafe` et
- un environnement d'exécution minimal.

## Rust dans un système embarqué sécurisé?

---

- Rust est basé sur LLVM  $\Rightarrow$  ARM est une cible possible.
- Chaîne de compilation facilitée via l'utilisation de `xargo`.

## Rust dans un système embarqué sécurisé?

---

- Rust est basé sur LLVM  $\Rightarrow$  ARM est une cible possible.
- Chaîne de compilation facilitée via l'utilisation de `xargo`.
- Depuis février 2018, création d'un groupe de travail pour les systèmes embarqués (*Embedded Devices Working Group*).

## Proportion de code

---

### ■ Système d'exploitation :

- ▶ 3279 lignes de code
- ▶ 473 lignes de code **unsafe**
- ▶ 15% de code **unsafe**
- ▶ 30 ko en -Os

## Proportion de code

---

- Système d'exploitation :
  - ▶ 3279 lignes de code
  - ▶ 473 lignes de code **unsafe**
  - ▶ 15% de code **unsafe**
  - ▶ 30 ko en -Os
- Environnement d'exécution Java Card
  - ▶ Travail en cours...

# Tester notre implémentation

---

- Tests sur l'ordinateur par praticité :
  - ▶ Reprise automatique des tests suite à un crash
  - ▶ Vérification de la procédure de portage à une autre architecture

## Tester notre implémentation

---

- Tests sur l'ordinateur par praticité :
  - ▶ Reprise automatique des tests suite à un crash
  - ▶ Vérification de la procédure de portage à une autre architecture
- Mais un ordinateur n'a pas d'UPM, et on veut tester notre modèle de sécurité.



## Tester notre implémentation

- Tests sur l'ordinateur par praticité :
  - ▶ Reprise automatique des tests suite à un crash
  - ▶ Vérification de la procédure de portage à une autre architecture
- Mais un ordinateur n'a **pas d'UPM**, et on veut tester **notre modele de sécurité**.

⇒ Émulation de l'UPM sur un Linux x86\_64

## Émulation de l'UPM

---

- L'UPM permet de protéger des blocs d'au moins 32 o
- `mprotect` ne permet de protéger que des blocs de 4 Kio... [Comment faire?](#)

## Émulation de l'UPM

---

- L'UPM permet de protéger des blocs d'au moins 32 o
- `mprotect` ne permet de protéger que des blocs de 4 Kio... [Comment faire?](#)
- Une erreur causée par `mprotect` cause un SIGSEGV
- `ptrace` permet d'attraper les SIGSEGV...

## Émulation de l'UPM

---

- L'UPM permet de protéger des blocs d'au moins 32 o
- `mprotect` ne permet de protéger que des blocs de 4 Kio... [Comment faire?](#)
- Une erreur causée par `mprotect` cause un SIGSEGV
- `ptrace` permet d'attraper les SIGSEGV...
- Il suffit d'avoir un exécutable qui `ptrace` l'exécutable qui effectue effectivement le test!
- Sur réception d'un SIGSEGV, on vérifie l'adresse mémoire accédée, et si elle est autorisée on libère la zone mémoire, on avance d'une instruction, et on re-verrouille la zone mémoire.
- Note : ce n'est pas un mécanisme dédié à la sécurité, juste aux tests.

## Émulation de l'UPM

- L'UPM permet de protéger des blocs d'au moins 32 o
- `mprotect` ne permet de protéger que des blocs de 4 Kio... [Comment faire?](#)
- Une erreur causée par `mprotect` cause un SIGSEGV
- `ptrace` permet d'attraper les SIGSEGV...
- Il suffit d'avoir un exécutable qui `ptrace` l'exécutable qui effectue effectivement le test!
- Sur réception d'un SIGSEGV, on vérifie l'adresse mémoire accédée, et si elle est autorisée on libère la zone mémoire, on avance d'une instruction, et on re-verrouille la zone mémoire.
- Note : ce n'est pas un mécanisme dédié à la sécurité, juste aux tests.
- Inconvénient : environ **un SIGSEGV et deux `mprotect` par accès mémoire**

## Quelques métriques

---

- Données sur la plate-forme :
  - ▶ Écriture en mémoire vive : 71 ns
  - ▶ Appel de fonction : 90 ns
  - ▶ Appel système : 2,6  $\mu$ s

## Quelques métriques

---

### ■ Données sur la plate-forme :

- ▶ Écriture en mémoire vive : 71 ns
- ▶ Appel de fonction : 90 ns
- ▶ Appel système : 2,6  $\mu$ s

### ■ Impact sur les performances :

- ▶ Accès matériel : un appel système
- ▶ Appel à une méthode d'un autre contexte d'exécution : 20  $\mu$ s, ie. environ 2 appels système et 200 écritures en mémoire vive.

## Conclusion

---

- Prototypage d'un système d'exploitation en Rust
- Amélioration de la sécurité des applications Java Card **via une UPM** :
  - ✓ sécurité en profondeur,
  - ✓ mécanisme de sécurité avec un surcoût très faible,
  - ⚠ la MVJC n'est plus générique (**mais peut fonctionner sans UPM**).
- Travail en cours sur l'environnement d'exécution Java Card.
  - ▶ Réflexion sur l'amélioration des mécanismes de sécurité.



## Conclusion

---

- Prototypage d'un système d'exploitation en Rust
- Amélioration de la sécurité des applications Java Card **via une UPM** :
  - ✓ sécurité en profondeur,
  - ✓ mécanisme de sécurité avec un surcoût très faible,
  - ⚠ la MVJC n'est plus générique (**mais peut fonctionner sans UPM**).
- Travail en cours sur l'environnement d'exécution Java Card.
  - ▶ Réflexion sur l'amélioration des mécanismes de sécurité.
- Ouverture des sources prévue...

# Questions ?

Guillaume BOUFFARD  
<guillaume.bouffard@ssi.gouv.fr>

Léo GASPARD  
<leo@gaspard.io>