OBERTHUR TECHNOLOGIES

De la Sécurité des Plateformes Java Card face aux Attaques Matérielles

Soutenance de thèse de Guillaume BARBU

Sous la direction de Philippe HOOGVORST, Guillaume DUC et Vincent GUERIN

MINES TELECOM INSTITUT Mines-Télécom

TELECOM ParisTech

EDiF

Agenda

- Carte à puce et Java Card
- Problématique de la sécurité des plateformes Java Card
- Attaques développées et mises en pratique
- Contremesures face à ces attaques
- Conclusions et perspectives

Agenda

- Carte à puce et Java Card

Carte à puce


optile
RATP
SNCF
17143170
BNP PARIBAS
4974
EXPIRE A FIN 03/14
M BARBU GUILLAUME
CARTE BLEUE
VISA
Vitale
carte d'assurance maladie
émise le 18/03/2010
GUILLAUME
BARBU
BORDEAUX maville
MONEO EXPIRE FIN 04/15
92500043300139
BNP PARIBAS
PREMIER
4974
EXPIRE A FIN 12/13
M. BARBU GUILLAUME
VISA
CANAL+
CANALSAT

Java Card

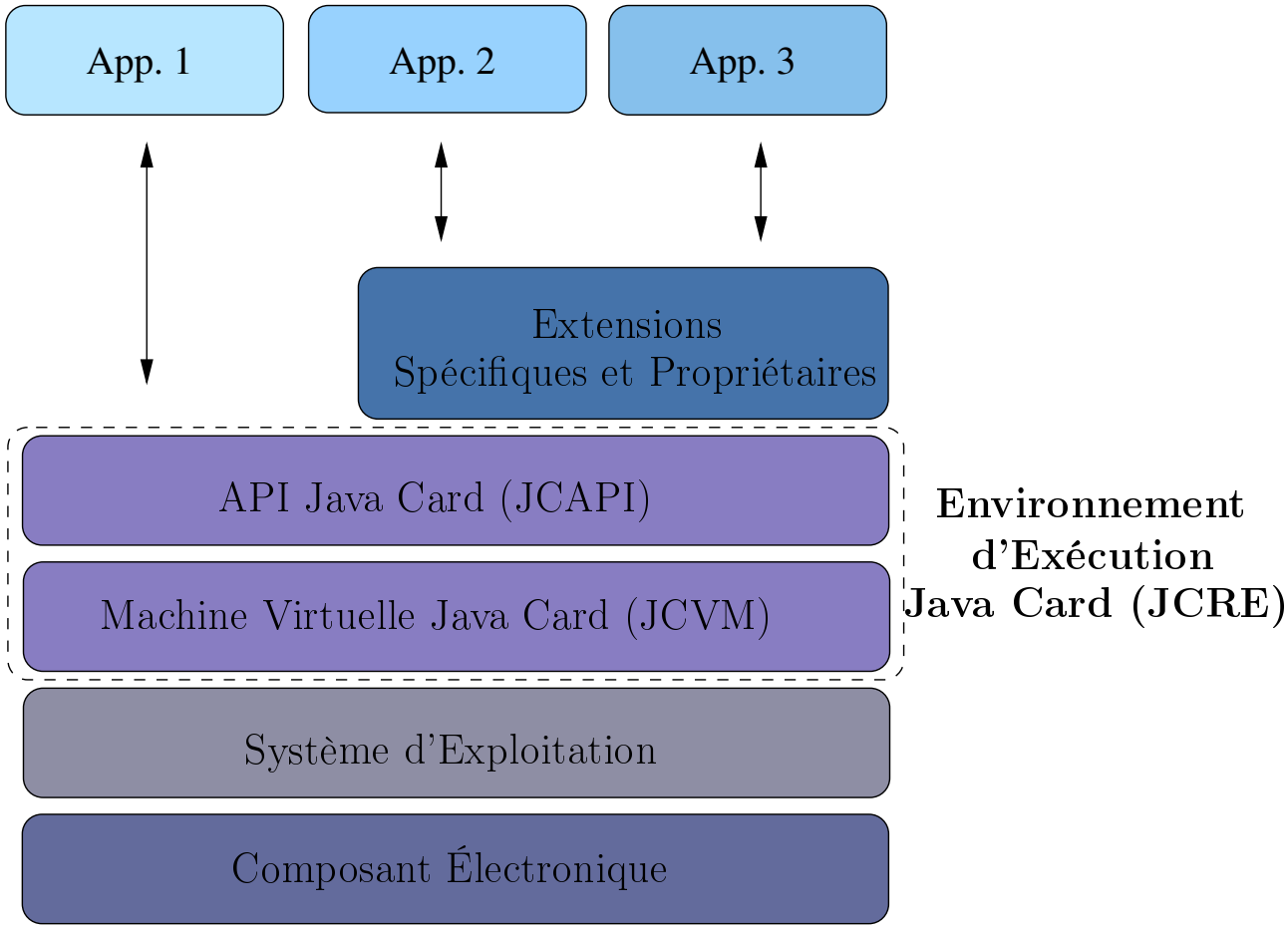
Introduite en 1996.

Vise à réduire les coûts de développement et de déploiement d'applications embarquées.

Définit une plateforme multi-applicative supportant le chargement d'applications post-issuance.

Est aujourd'hui la technologie leader sur le marché des cartes à puce.

Architecture de la plateforme



Agenda

- Problématique de la sécurité des plateformes Java Card
  - Les attaques logicielles
  - Les attaques matérielles

La sécurité des Java Cards

Initialement, sont considérées séparément :

La problématique venant de la possibilité de chargement d'applications, potentiellement malicieuses

- i.e. les attaques logiques.

La problématique venant de la possibilité de perturber le composant électronique

- i.e. les attaques par injection de faute.

Contre les attaques logicielles

Les contraintes du langage Java

- dont la chaîne de compilation/conversion ainsi que le ***byte-code verifier*** assurent le respect.

Les mécanismes sécuritaires internes de la plateforme

- pare-feu inter-applications, vérification des liens, transactions.

Nombre d'attaques dans la littérature considèrent néanmoins qu'une application mal-formée (i.e. ne passant pas le bytecode verifier) peut être chargée sur une carte.

Contre les attaques matérielles

Les mécanismes sécuritaires du composant électronique

- conception, détecteurs/capteurs.

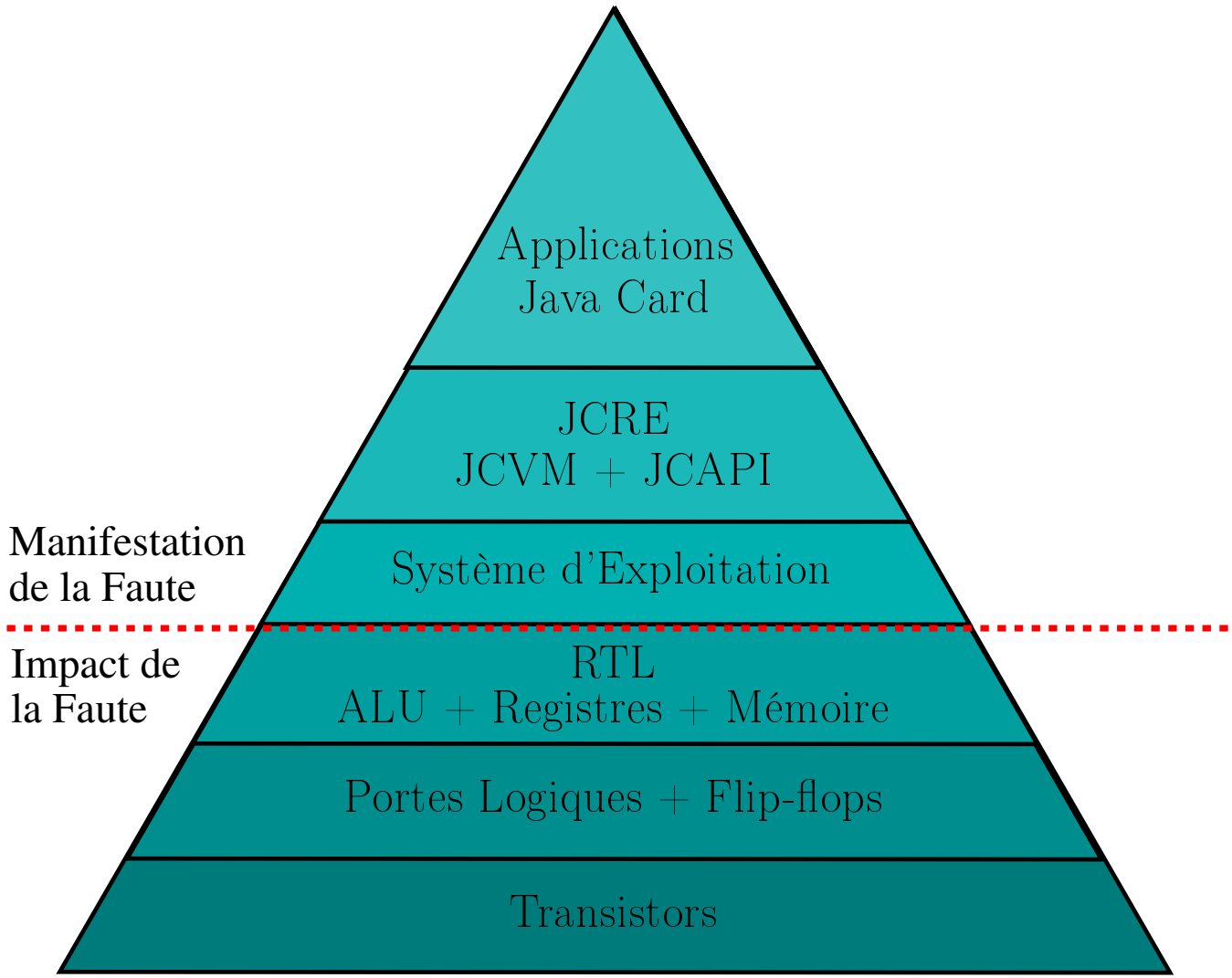
Des contremesures logicielles

- principalement au niveau des instructions de branchement conditionnel et des valeurs testées.

Agenda

- Attaques développées et mises en pratique
  - Introduction des attaques combinées (CA)
  - Un exemple détaillé : CA par confusion de type
  - Autres CA développées

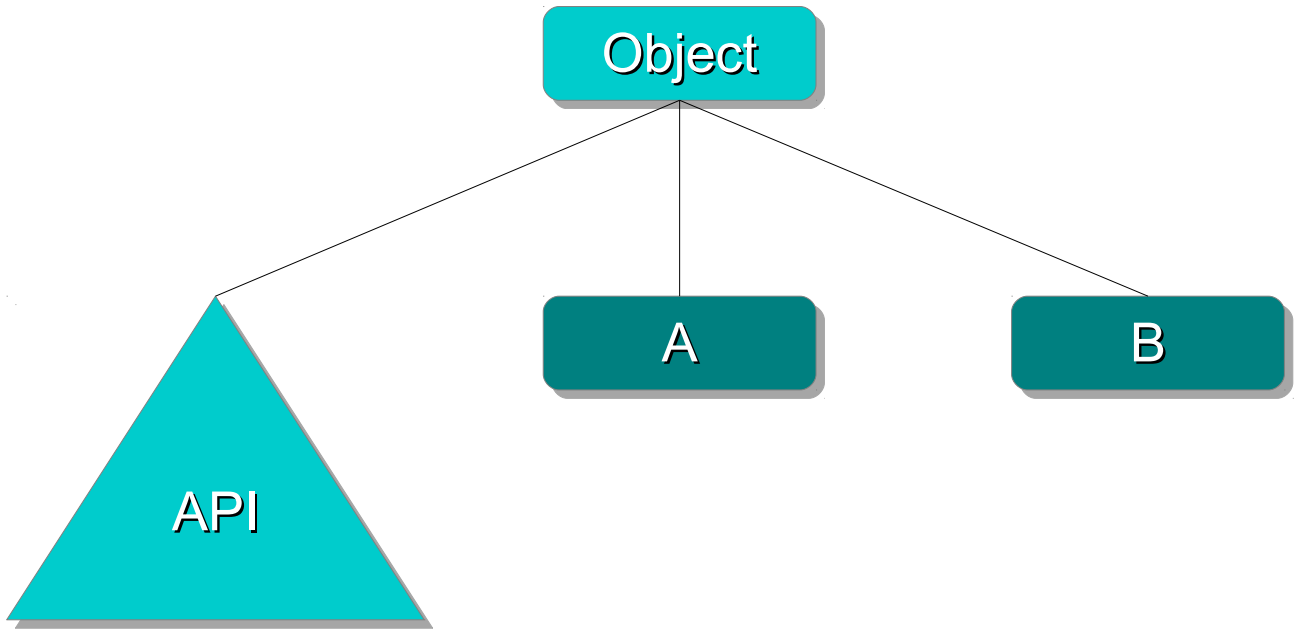
Introduction des attaques combinées (CA)



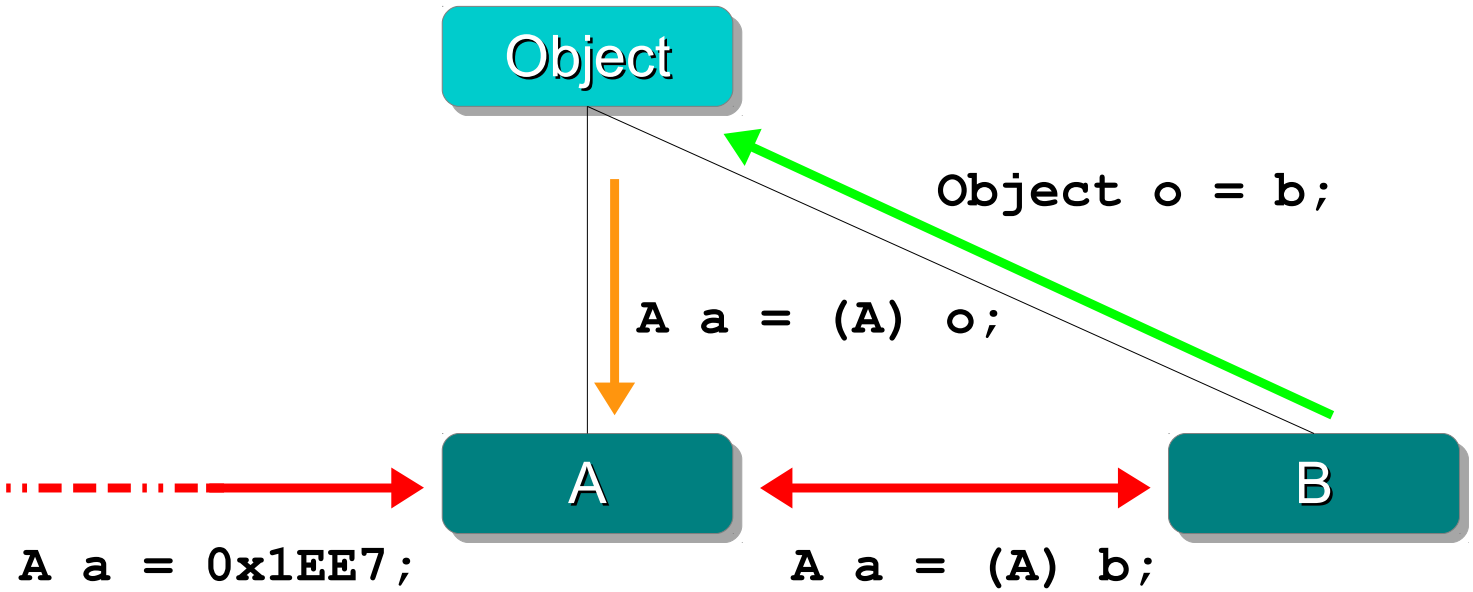
CA par confusion de type [BTG10]

Les types Java :

- Primitifs (byte, short, int, ...) ;
- Classes.

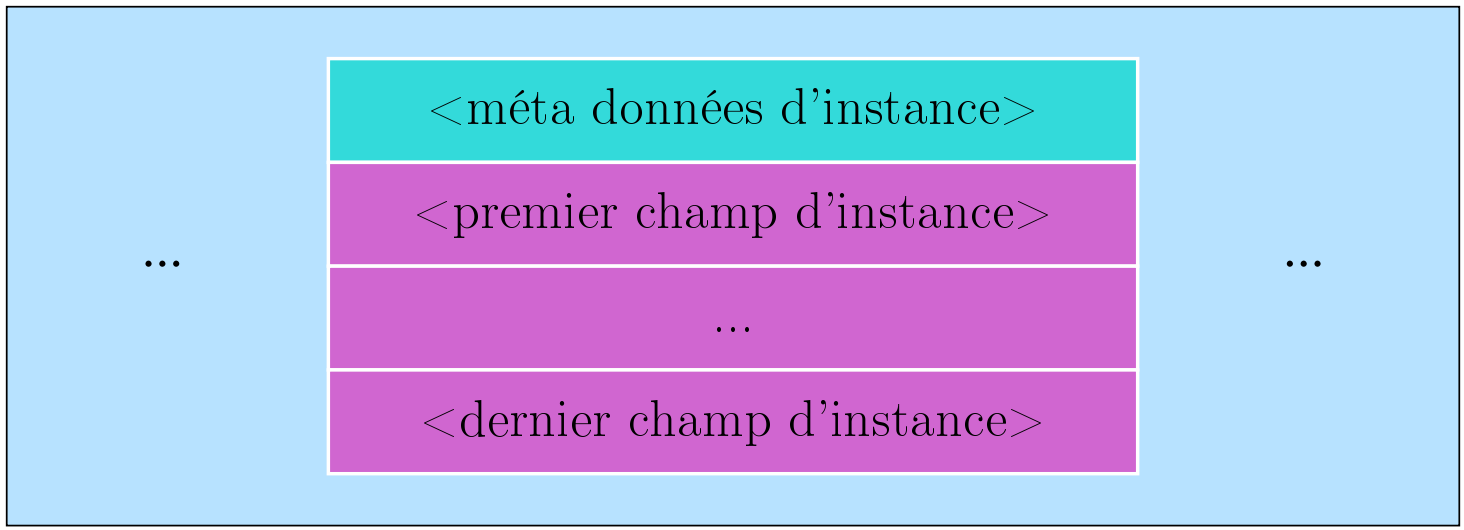


CA par confusion de type [BTG10]



CA par confusion de type [BTG10]

Hypothèse sur l'organisation de la mémoire contenant les instances de classes (le tas Java)

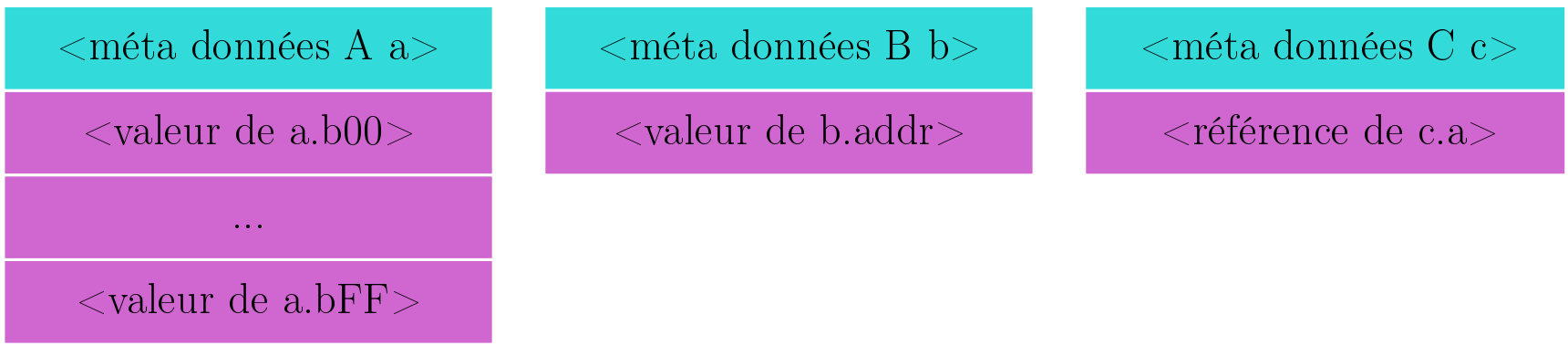


CA par confusion de type [BTG10]

Les classes dans l'application de l'attaquant :

```
public class A { byte b00, b01, ..., bFF ; }
public class B { short addr ; }
public class C { A a ; }
```

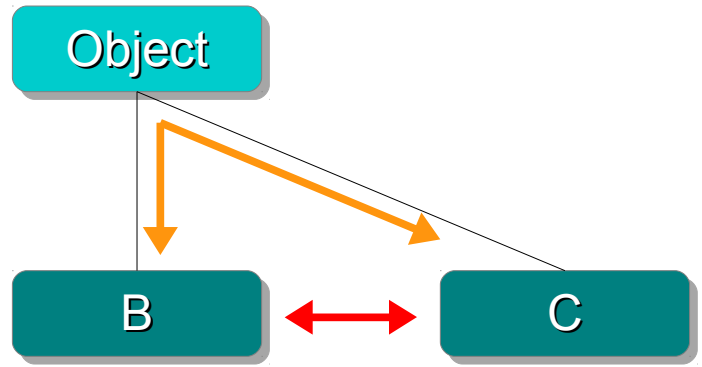
Dans le tas Java :



CA par confusion de type [BTG10]

Le code dans l'application de l'attaquant :

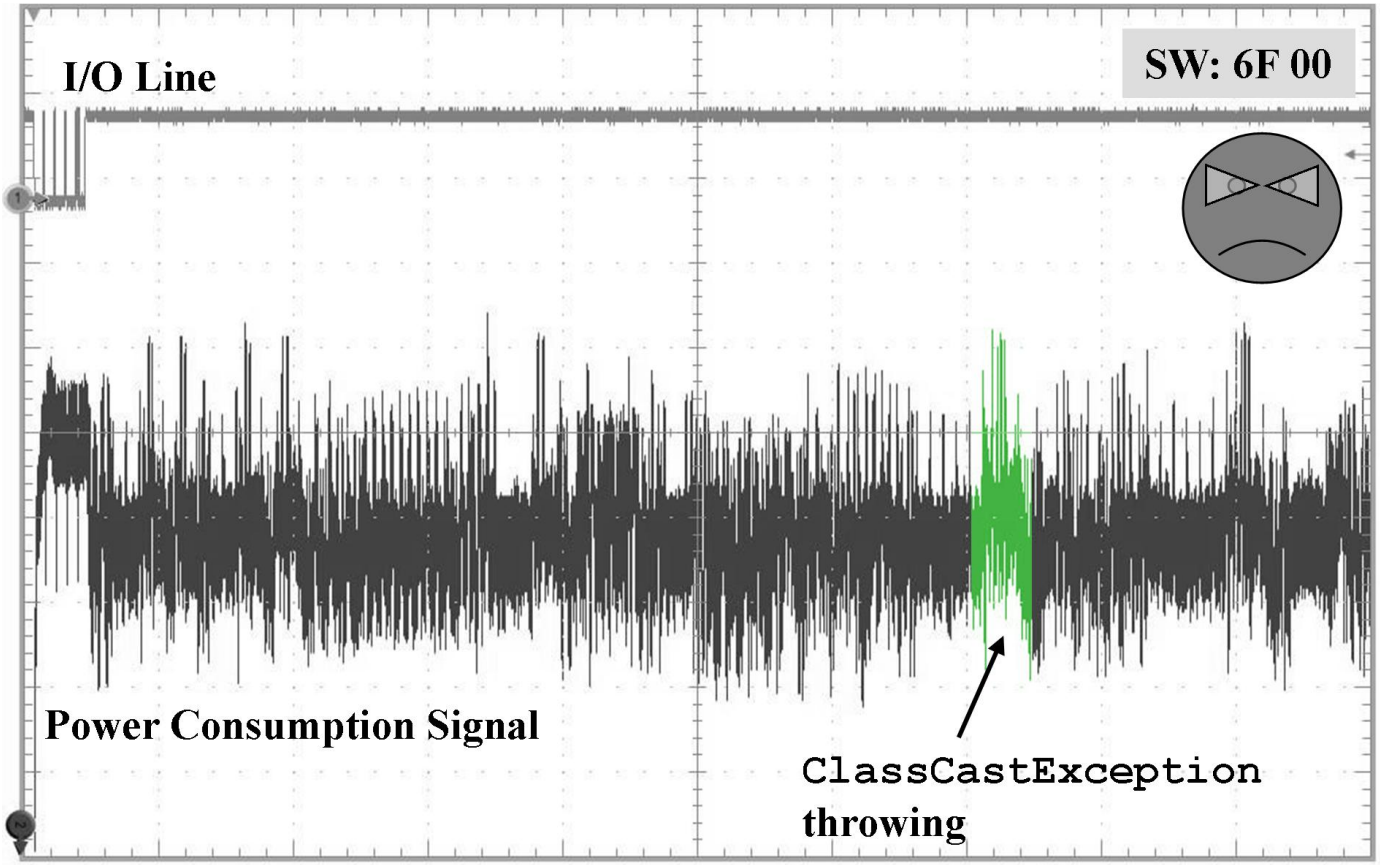
- `B b = (B) (Object) c;`



Le transtypage est vérifié à l'exécution de l'instruction `checkcast` ➔ `ClassCastException`.

Cette opération est visible en analysant la consommation de courant de la carte durant l'exécution.

CA par confusion de type [BTG10]



CA par confusion de type [BTG10]

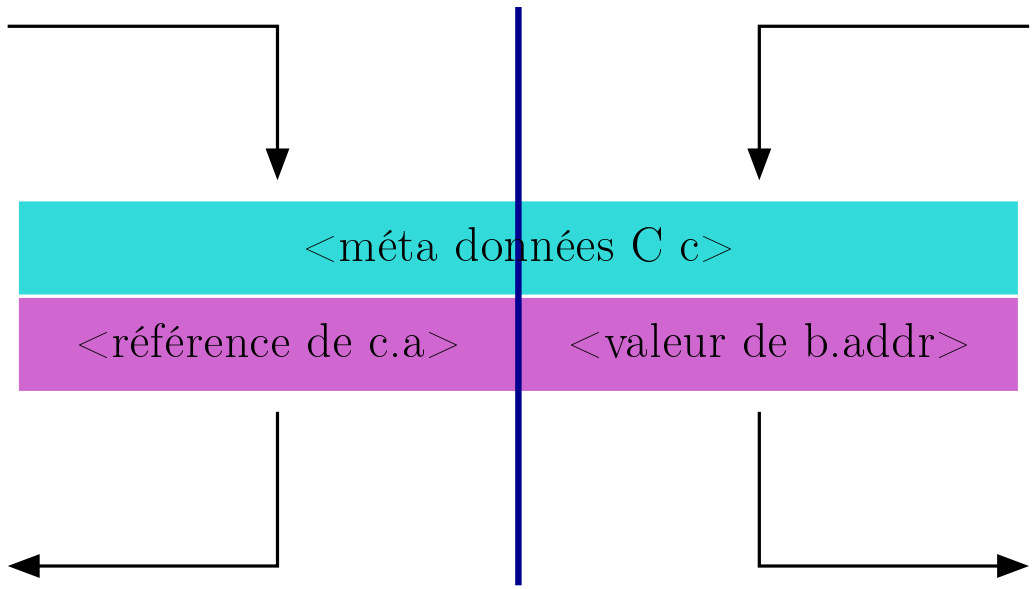
Le but de l'attaque par injection de faute est de permettre l'opération de transtypage.

Pour cela, l'attaquant peut perturber l'exécution de l'instruction checkcast pendant le test de validité du transtypage.

Expérimentalement, nous avons pu réaliser la perturbation voulue sur un banc d'attaque laser.

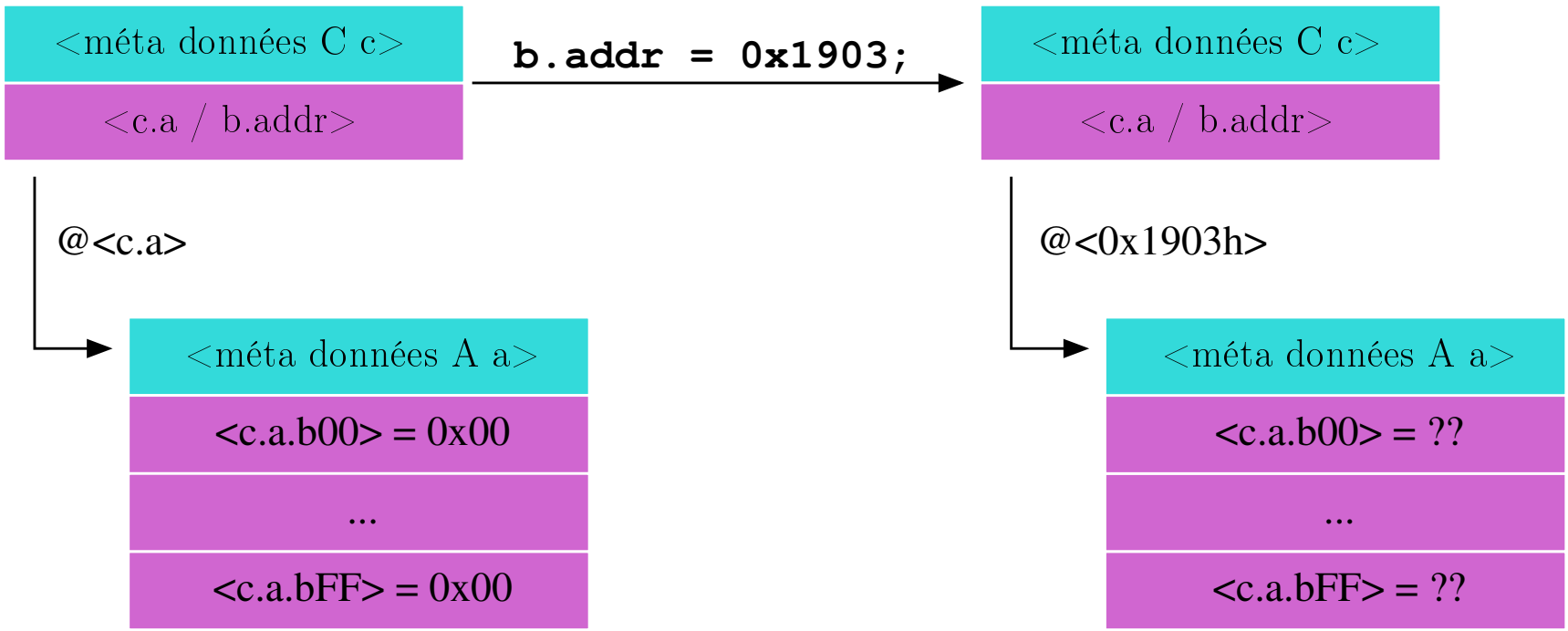
CA par confusion de type [BTG10]

Une fois l'attaque par faute réussie, l'attaquant a accès à un objet de type `C` soit en tant qu'instance de la classe `B`, soit en tant qu'instance de la classe `C`.



CA par confusion de type [BTG10]

L'attaquant est donc capable de forger la référence d'un objet de type `A`.



CA par confusion de type [BTG10]

Exploitation sur une plateforme Java Card 3 *Connected Edition*.

```
String name = b2String(buf, ISO7816.OFFSET_CDATA);
while (!classFound) {
  try {
    b.addr++; // incrément la référence de c.a
    Object o = (Object) (c.a);
    if ( o instanceof Class )
      if ( ((Class) o).getName().equals(name))
        classFound = true ;
  } catch (SecurityException se){}
}
```

CA par confusion de type [BTG10]

Une fois la classe ciblée trouvée, l'attaquant peut alors modifier son contenu et notamment son tableau d'instructions.

Que cette classe appartienne à sa propre application ou non.

Il peut donc à sa guise :

- Modifier le code de sa propre application ;
- Modifier le code d'autres applications.

D'autres CA

[BT11]
- Fixer un point d'arrêt dans une application embarquée.
- En modifiant le pointeur de données d'un `byte[]`.
- Déni de service contre l'ordonnanceur de *thread*.

[BDH11]
- S'authentifier indûment.
- En perturbant un opérande empilé.
- En utilisant l'interface `Authenticator`

D’autres CA

[BHD12a]

- Contourner le pare-feu inter-applications par rejeu d’application.
- En prédisant les références qui sont affectées.
- En utilisant le mécanisme de ramasse-miettes et une faiblesse des dernières spécifications.

[BGG12]

- Espionner des communications sécurisées.
- Obtenir des privilèges dans différents environnements.
- En stockant la référence du buffer APDU.
- En utilisant la notion de *restartable task* et d’*event*.

D’autres CA

[BHD12b]

- Créer une confusion de type.
- Perturber le déroulement d’une application.
- En utilisant différentes exceptions.
- En perturbant les mécanismes liés à la gestion des exceptions.

Agenda

- Contremesures face à ces attaques
  - Dans une machine virtuelle *défensive*
  - Dans une machine virtuelle *défendante*

La machine virtuelle *défensive*

Machine virtuelle *défensive*

Une machine virtuelle Java Card est dite *défensive* si sa sécurité repose sur la vérification des applications chargées sur la carte par un bytecode verifier embarqué.

Dans ce contexte, la plateforme ne protège pas *a priori* les applications qu'elle contient.

Ces applications se doivent donc d'assurer leur propre sécurité.

Outil de sécurisation statique des applications

Les instructions conditionnelles `ifeq` et `ifne` sont des cibles toutes désignées pour des attaques en faute.

Le brevet [FR1158517] expose un procédé permettant de sécuriser les instructions conditionnelles contre des attaques par injection(s) de faute(s) simple et double.

Outil de sécurisation statique des applications

Au lieu d'ajouter des contrôles redondants au sein du code source.

```
if (cond) {


   processTrue();
}
else                {


   processFalse();
}
```

Outil de sécurisation statique des applications

Au lieu d'ajouter des contrôles redondants au sein du code source.

```
if (cond) {
  if (!cond)
    ISOException.throwIt(SW_SECURITY);
  else
    processTrue();
}
else if (!cond) {
  if (cond)
    ISOException.throwIt(SW_SECURITY);
  else
    processFalse();
}
```

Outil de sécurisation statique des applications

Le procédé proposé consiste à effectuer des modifications au niveau du bytecode dans le fichier .CLASS.

```
    iload 4          // push cond onto the operand stack

    ifeq L1          // branch at L1 if cond is 0 (false)




L2: ...              // processTrue
    goto L3
L1:
    ...              // processFalse
L3: ...
```

Outil de sécurisation statique des applications

Le procédé proposé consiste à effectuer des modifications au niveau du bytecode dans le fichier .CLASS.

```
     iload 4          // push cond onto the operand stack
     iload 4          // push cond onto the operand stack
     ifeq L1          // branch at L1 if cond is 0 (false)
     iconst_1         // push 1 onto the operand stack
     if_icmpeq L2     // branch at L2 if cond equals 1 (true)
L4:  ...              // fault detected raise exception
     goto L3
L2:  ...              // processTrue
     goto L3
L1:  ifne L4          // branch at L4 to handle fault
     ...              // processFalse
L3:  ...
```

Outil de sécurisation statique des applications

Le code généré automatiquement résiste à des attaques simples et doubles en considérant les deux modèles de faute généralement admis (saut d'instructions et modification de données).

La taille du code supplémentaire est diminuée de moitié.

Le code généré respecte toujours les spécifications et pourra être utilisé sur n'importe quelle VM avec le même niveau de sécurité.

La machine virtuelle *défendante*

Machine virtuelle *défendante*

Une machine virtuelle Java Card est dite *défendante* si sa sécurité repose sur des vérifications qu'elle réalise durant l'exécution des applications.

Dans ce contexte, la plateforme se protège et protège dans une certaine mesure les applications qu'elle contient.

Contrôle d'intégrité de la pile d'opérande

Une des attaques décrites précédemment tire partie d'une faute injectée sur un opérande.

Nous avons donc proposé et comparé différentes méthodes permettant d'assurer l'intégrité de la pile d'opérande.

La méthode la plus efficace repose sur l'introduction d'une variable associée à une *frame* Java permettant d'exhiber une propriété invariante.

Contrôle d'intégrité de la pile d'opérande

Propriété invariante

Soit σ la somme (au sens de l'opérateur XOR) de toutes les valeurs empilées et dépilées au sein d'une frame et Σ_t la somme des valeurs sur la pile d'opérande à un instant t.
Alors $\forall t, \Sigma_t \oplus \sigma = 0$

Conserver σ à jour nécessite donc de modifier les routines d'empilement et de dépilement au sein de la machine virtuelle.

La vérification de l'invariant peut être opérée lors des contrôles du pare-feu et lors des ruptures du flot de contrôle.

Contrôle d'intégrité de la pile d'opérande

Instructions	Coût
`aload+astore`	+ 12,29 %
`aload+getfield+astore`	+ 11,75 %
`aload+aload+putfield`	+ 17,59 %
`aload+invokevirtual+return`	+ 1,77 %
`aload+invokevirtual+areturn+astore`	+ 2,38 %
`getstatic+astore`	+ 10,21 %

Autres réponses dans une VM *défendante*

Une VM défendante est particulièrement adaptée pour contrer des attaques combinées.

En particulier, les attaques décrites peuvent toutes être mises en échec par des vérifications adéquates.

Nous avons également décrit une méthode de modification aléatoire du jeu d'instructions afin de compliquer la tâche d'un attaquant voulant faire exécuter du code injecté. [BA12]

Agenda

- Conclusions et perspectives

Conclusions

Les travaux entrepris durant cette thèse ont permis de présenter les premières attaques combinées sur des Java Cards.

Les CA montrent qu'il n'est pas nécessaire de charger du code mal-formé pour exécuter du code mal-formé.

Mais elles permettent également de construire des attaques plus puissantes que les seules attaques logiques ou physiques.

Perspectives

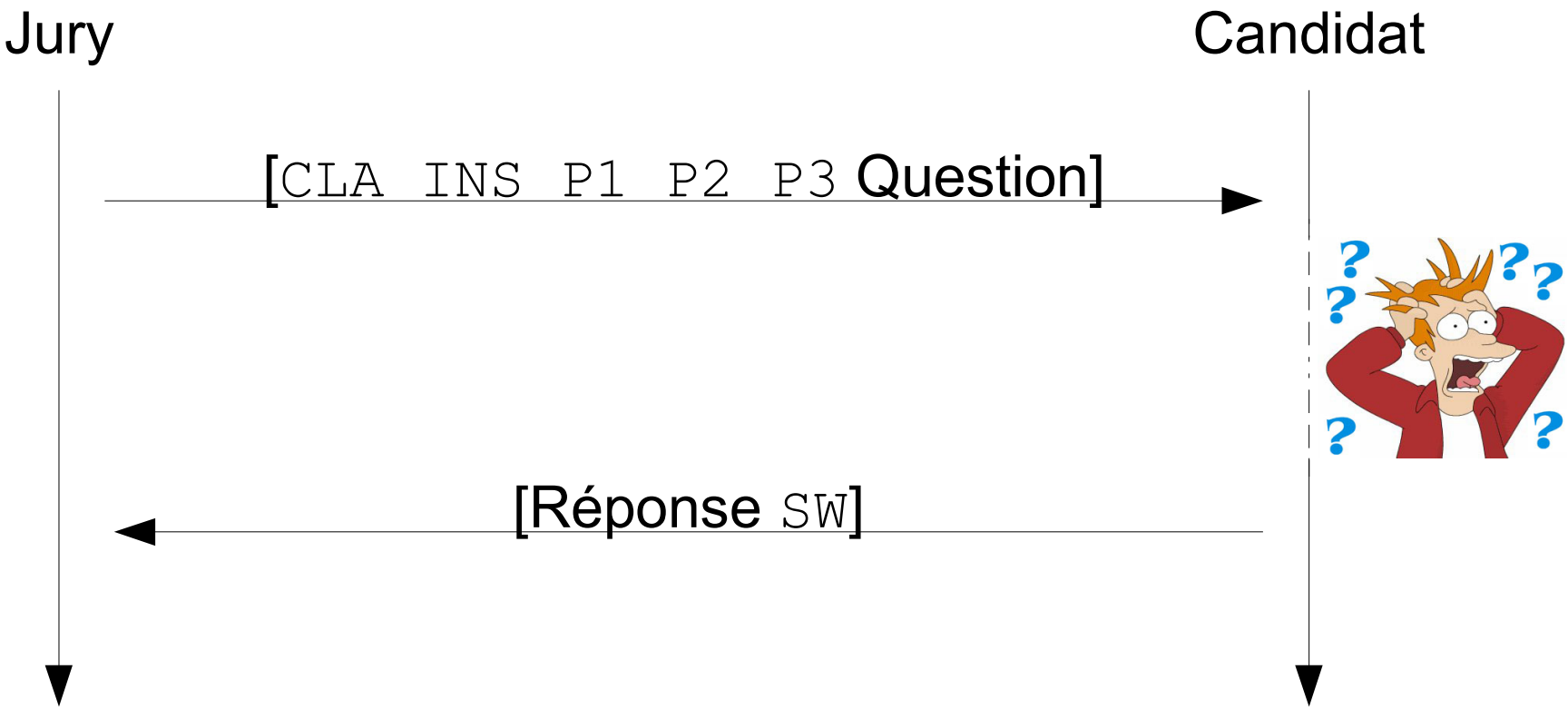
Autres travaux en cours sur la sécurité des Java Cards

- VM reconnaissant plusieurs jeux d'instructions aléatoires. [Brevet en cours de dépôt]
- Activation et inhibition dynamique par la VM des contremesures aux attaques par injection(s) de faute(s). [Brevet en cours de dépôt, Article soumis à CARDIS 2012]

Recherche de mécanismes performants pour une VM défendante, notamment pour assurer la sécurité du typage.

Possibilité de porter les CA sur d'autres plateformes (embarquées).

Merci pour votre attention




Guillaume Barbu and Philippe Andouard.
Instruction-Set Randomization on Java Card Platforms.
In *Chip-to-Cloud Security Forum*, 2012.


Guillaume Barbu, Guillaume Duc, and Philippe Hoogvorst.
Java Card Operand Stack : Fault Attacks, Combined Attacks and Countermeasures.
In Prouff [Pro11], pages 297–313.


Guillaume Barbu, Christophe Giraud, and Vincent Guerin.
Embedded Eavesdropping on Java Card.
In D. Gritzalis, S. Furnell, and M. Theoharidou, editors, *International Information Security and Privacy Conference – SEC 2012*, volume 376 of *IFIP Advances in Information and Communication Technology*, pages 37–48. Springer, 2012.


Guillaume Barbu, Philippe Hoogvorst, and Guillaume Duc.

Application-Replay Attack on Java Cards : When the Garbage Collector Gets Confused.
In G. Barthe and B. Livshits, editors, *International Symposium on Engineering Secure Software and Systems – ESSoS 2012*, volume 7159 of *LNCS*. Springer, 2012.

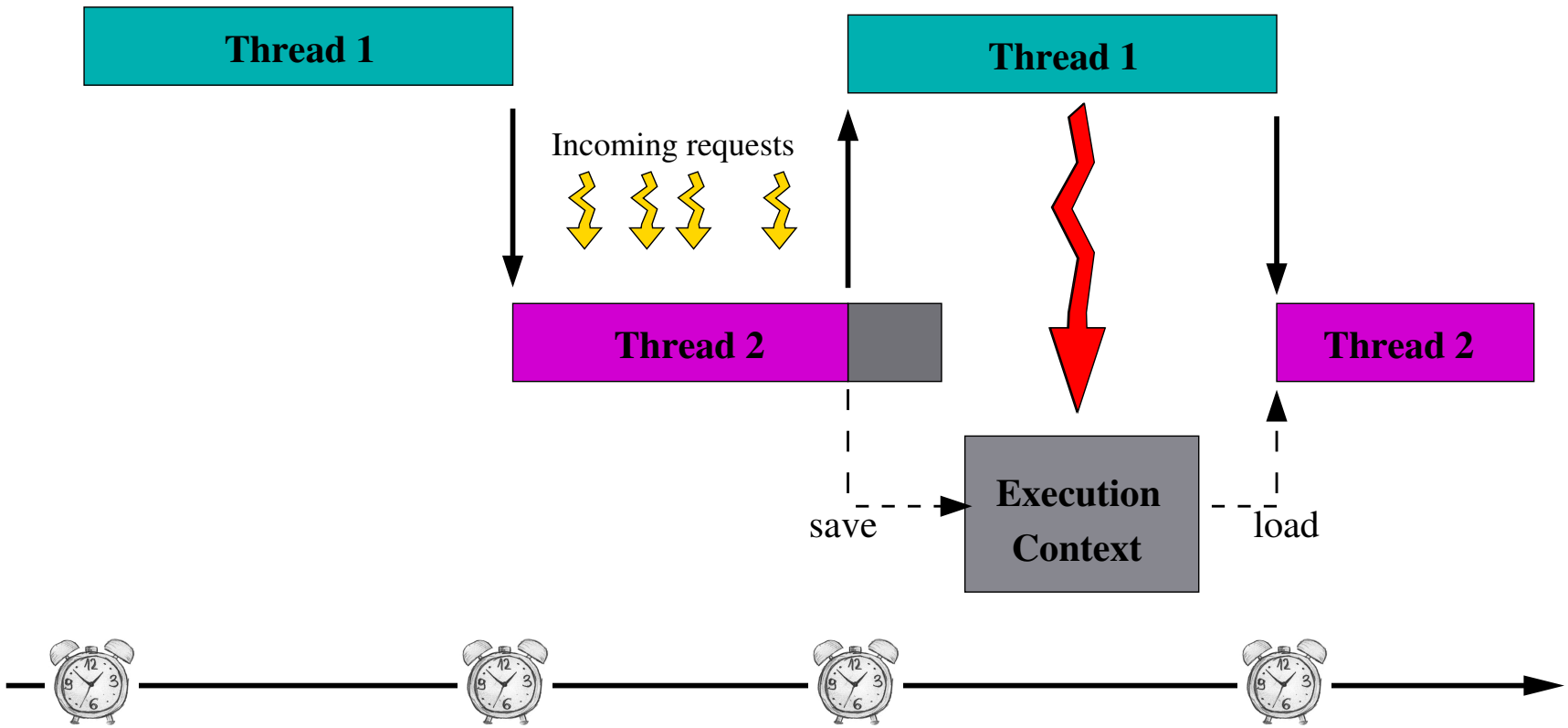

Guillaume Barbu, Philippe Hoogvorst, and Guillaume Duc.
Tampering with Java Card Exceptions : the `Exception` Proves the Rule.
In *International Conference on Security and Cryptography – SECRYPT 2012*. SciTePress Digital Library, 2012.

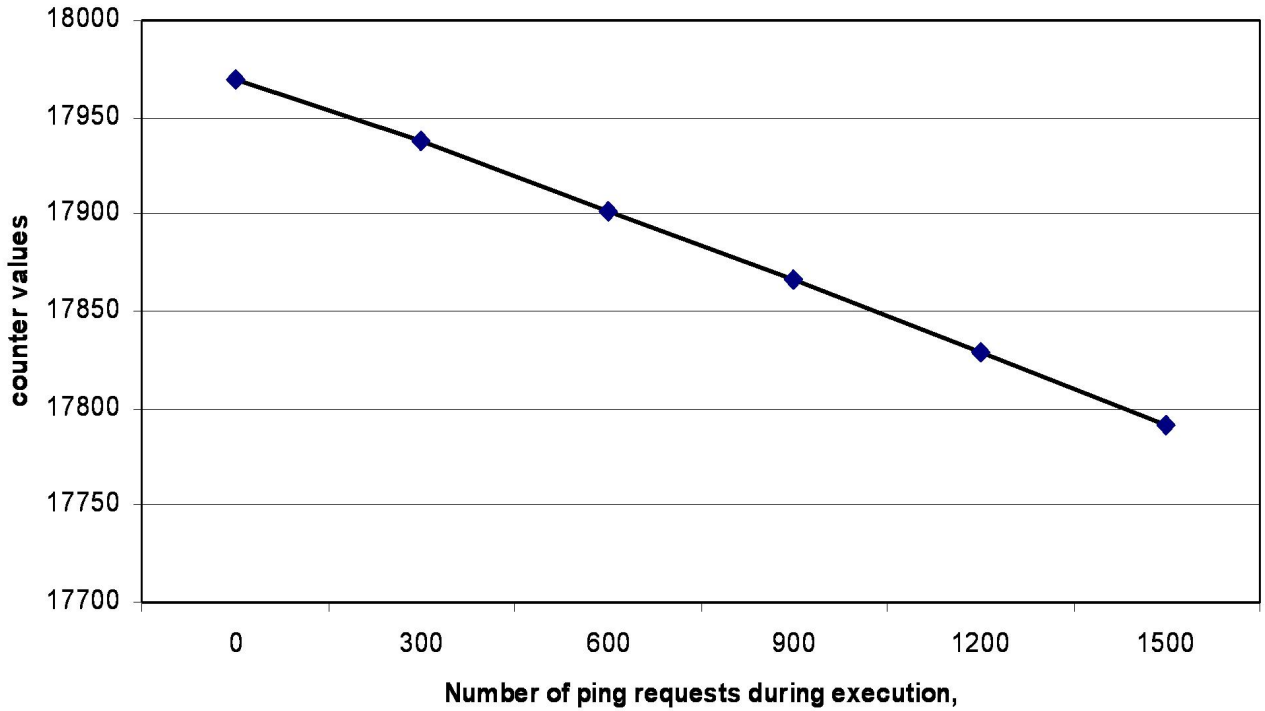

Guillaume Barbu and Hugues Thiebeauld.
Synchronized Attacks on Multithreaded Systems - Application to Java Card 3.0 -.
In Prouff [Pro11], pages 18–33.

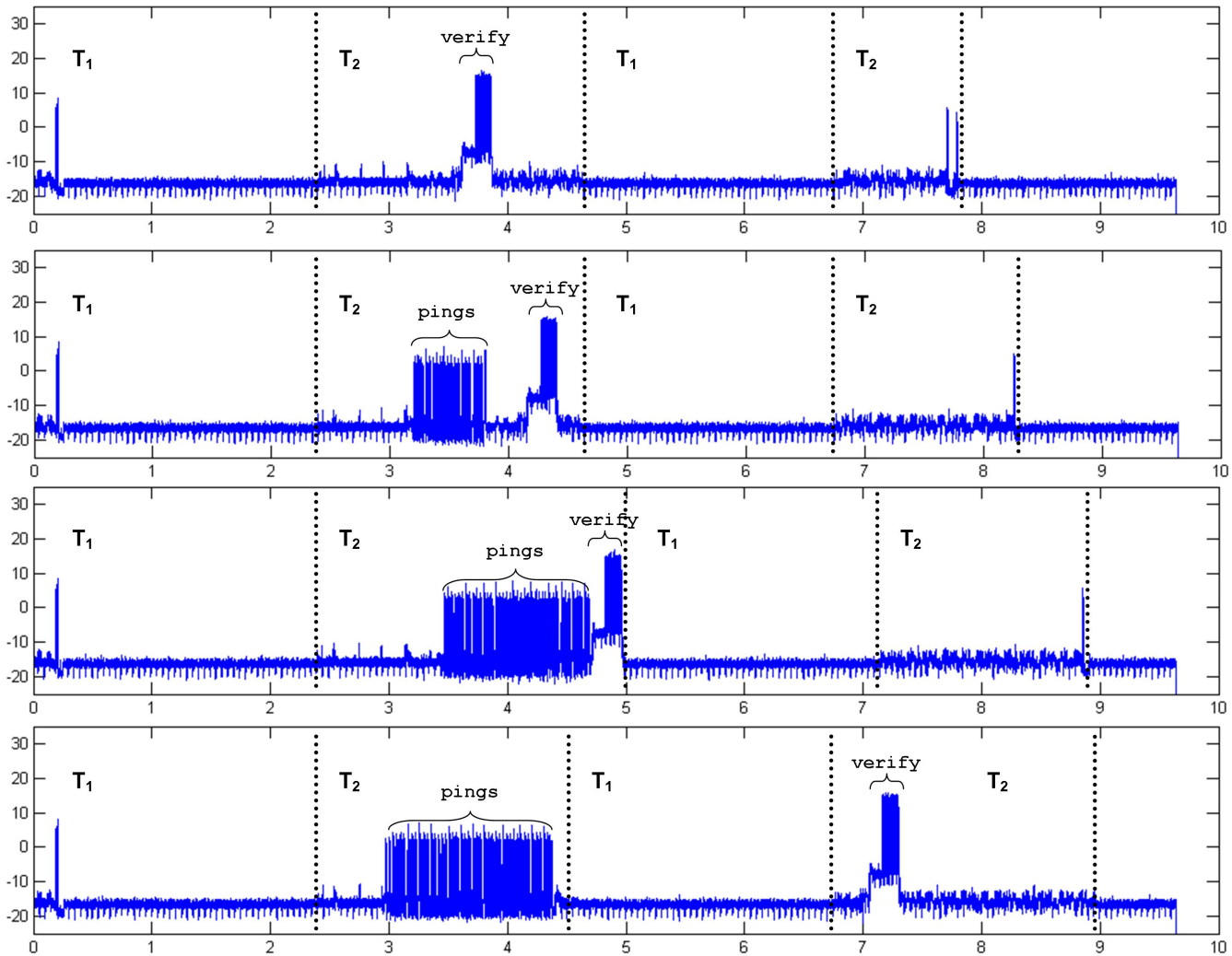

Guillaume Barbu, Hugues Thiebeauld, and Vincent Guerin.

Attacks on Java Card 3.0 Combining Fault and Logical Attacks.
In Dieter Gollmann and Jean-Louis Lanet, editors, *Smart Card Research and Advanced Applications, 9th International Conference – CARDIS 2010*, volume 6035 of *LNCS*, pages 148–163. Springer, 2010.

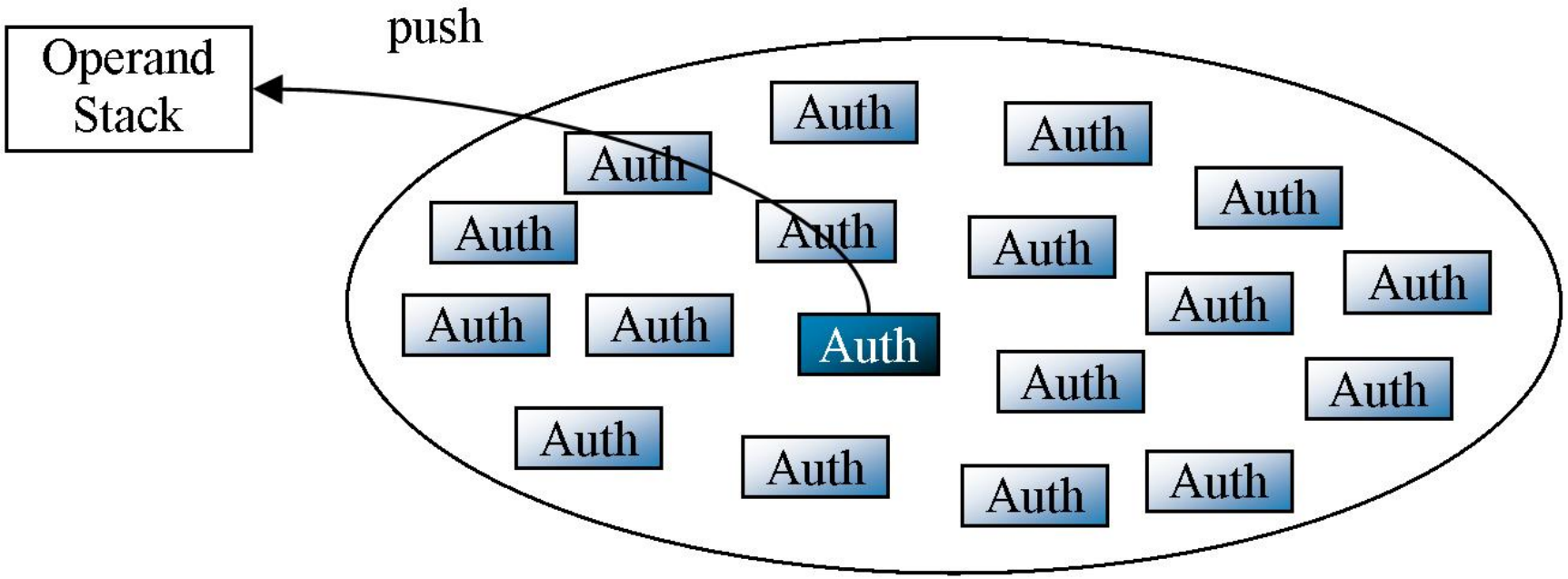
Emmanuel Prouff, editor.
Smart Card Research and Advanced Applications, 10th International Conference – CARDIS 2011, volume 7079 of *LNCS*. Springer, 2011.

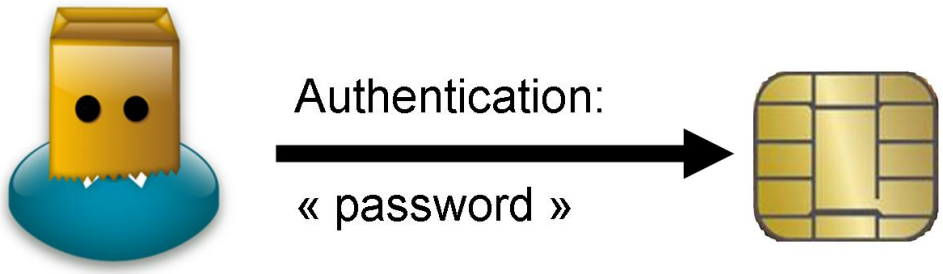

Thread 1
Thread 1
Incoming requests
Thread 2
Thread 2
save
Execution Context
load




T1
T2
verify
T1
T2
T1
T2
pings
verify
T1
T2
pings
verify
T1
T2
T1
T2
pings
verify

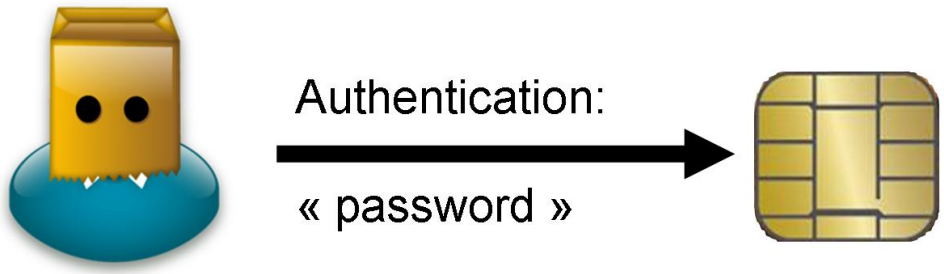
[BDH11]


Operand Stack
push
Auth
Auth
Auth
Auth
Auth
Auth
Auth
Auth
Auth
Auth
Auth
Auth
Auth
Auth
Auth
Auth
Auth
Auth
Auth




```
if (! auth.check (submitted))
        return ACCESS_DENIED;
else
        return ACCESS_GRANTED;
```

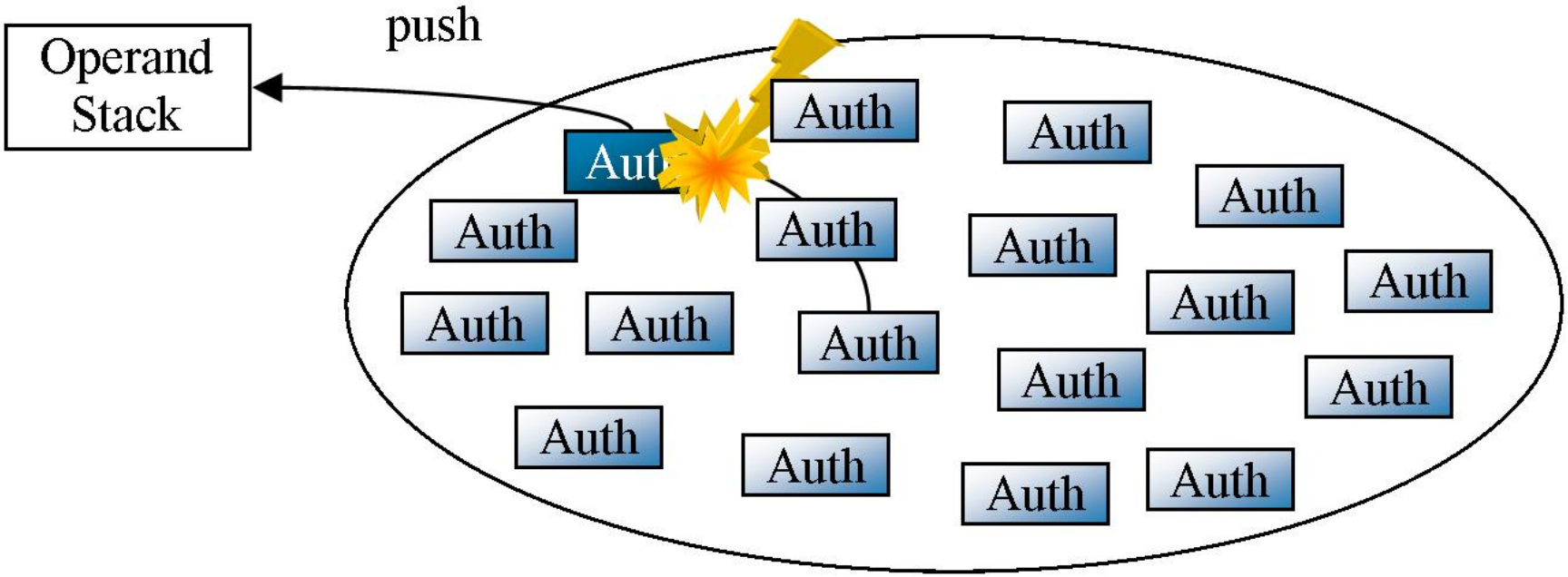
```
public boolean check (String password) {
        // PASSWORD = « p@ssw0rd »
        return secureCompareString (password, PASSWORD);
}
```

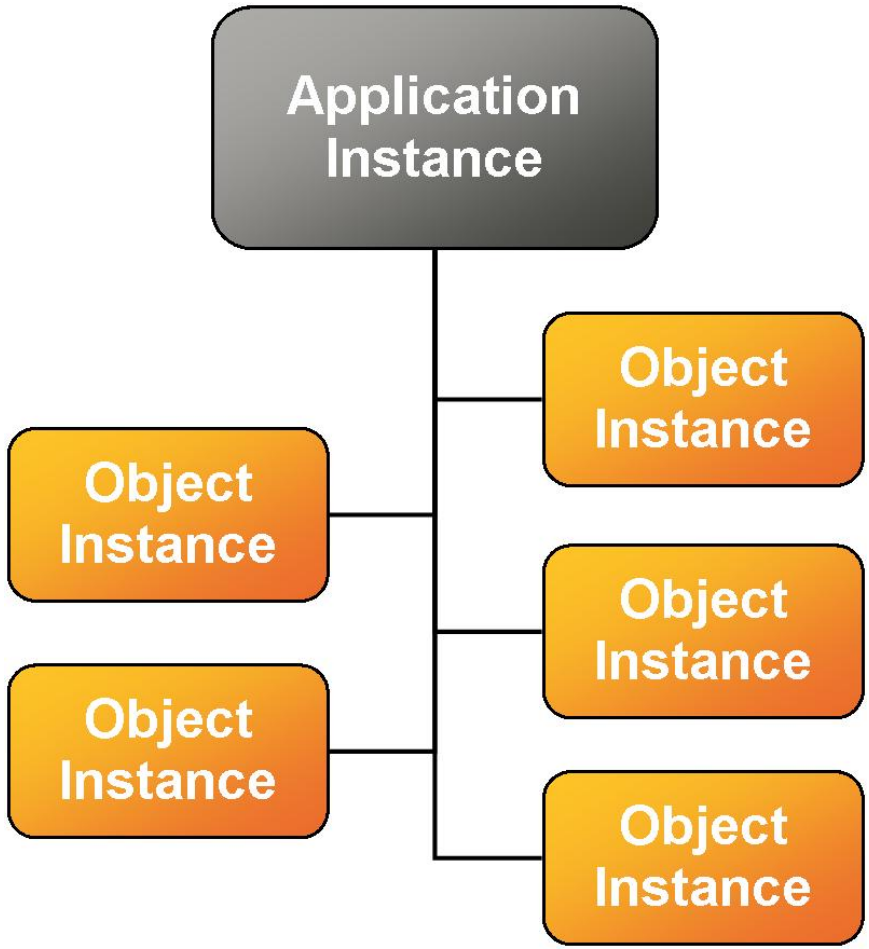


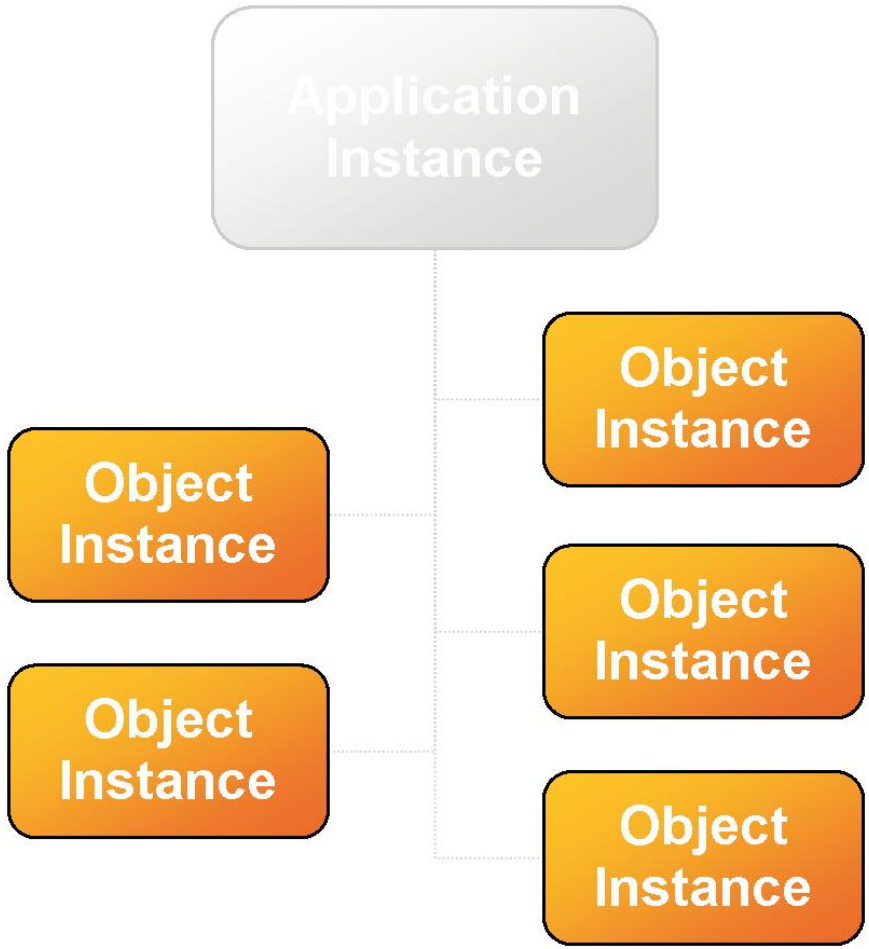

```
if (! auth.check (submitted))
        return ACCESS_DENIED;
else
        return ACCESS_GRANTED;
```

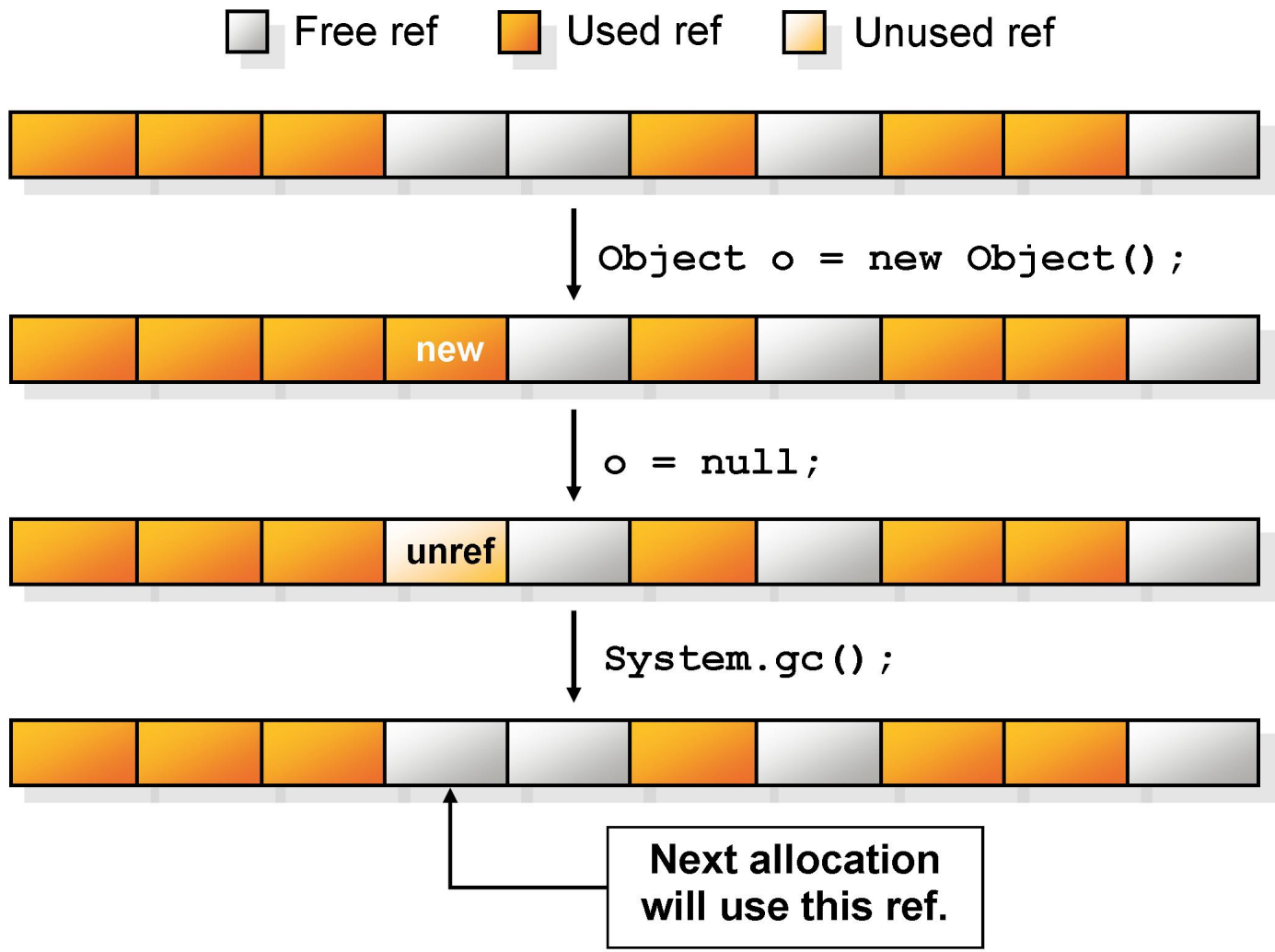
```
public boolean check (String password) {
        return true;
}
```

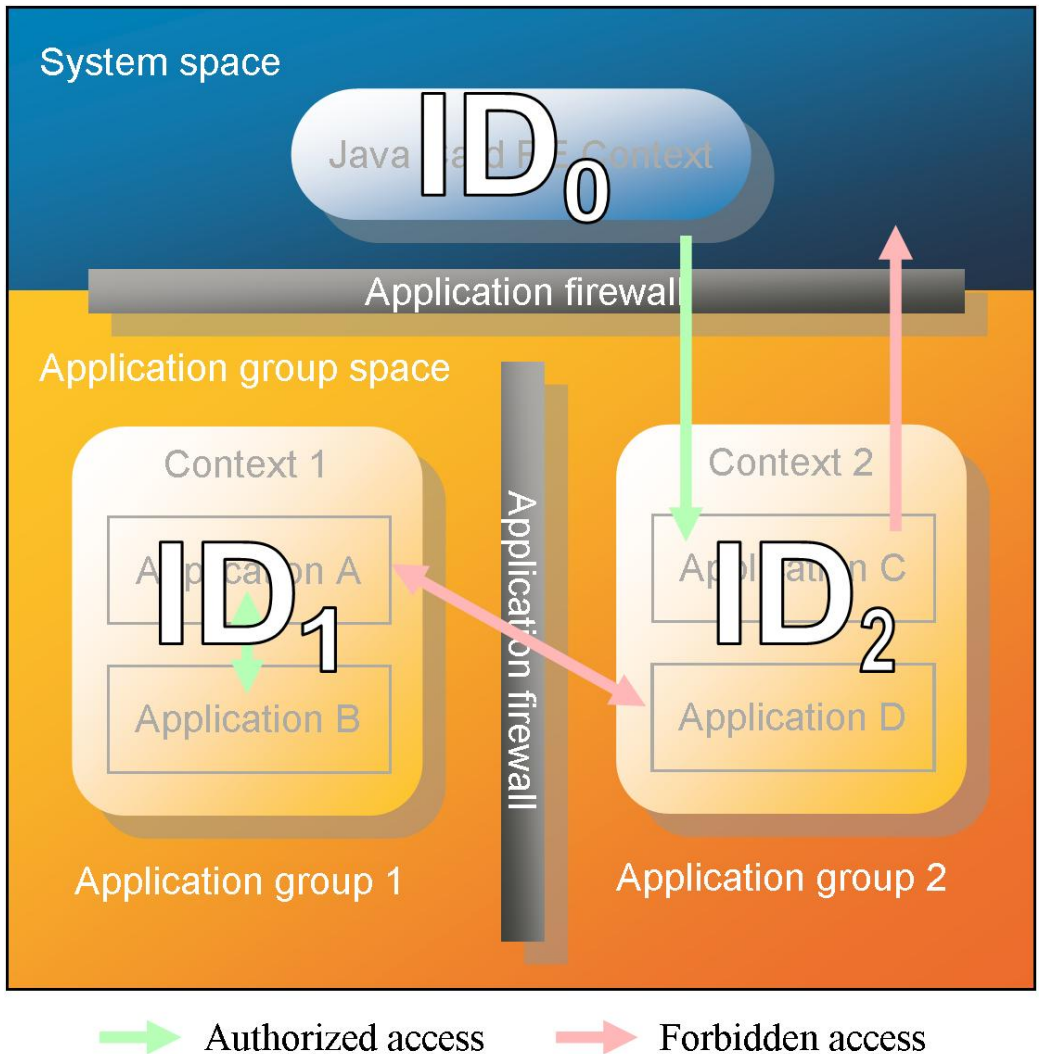
[BDH11]




Application Instance
Object Instance
Object Instance
Object Instance
Object Instance
Object Instance


Application Instance
Object Instance
Object Instance
Object Instance
Object Instance
Object Instance
Object Instance


Free ref
Used ref
Unused ref
Object o = new Object();
new
o = null;
unref
System.gc();
Next allocation will use this ref.


System space
Java Card RE Context
ID0
Application firewall
Application group space
Context 1
Application A
Application B
ID1
Application firewall
Context 2
Application C
Application D
ID2
Application group 1
Application group 2
Authorized access
Forbidden access

Global Arrays

Un global array est un type de tableau particulier qui appartient au JCRE, mais qui est accessible par toutes les appliations.

Extrait du §2.4.2.8 du JCRE 3.0.1

"Accessing Class Instance Object Fields (...). Otherwise, if the bytecode is `putfield` *and the field being stored is a reference type and the reference being stored is a reference to a temporary JCRE entry point object or a global array, access is denied."*

Dans le contexte de l'environnement GlobalPlatform

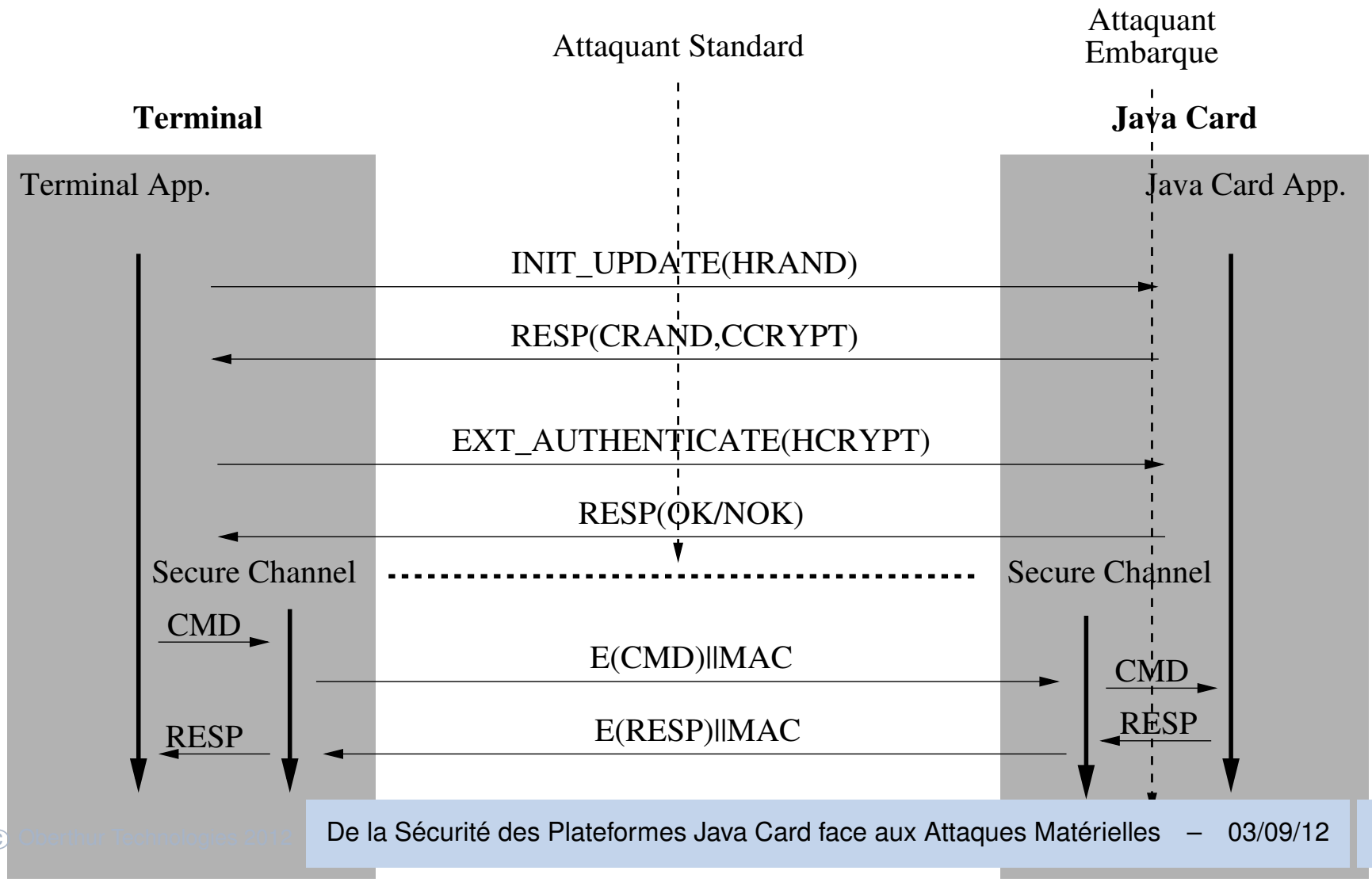
```
public class MyApplet extends Applet,
                      implements CLApplet {
    ...
    public void notifyCLEvent(short event) {
        analyseAPDU(); // en utilisant la référence stockée
    }
}
```

Dans le contexte du CAT/(U)SAT

```
public class MyApplet    extends Applet,
                         implements ToolkitInterface {
  ToolkitRegistry r;
  public MyApplet() {
    r = ToolkitRegistrySystem.getEntry();
    ...
    r.setEvent(ToolkitConstants.EVENT_UNFORMATTED_SMS_PP_UPD);
  }

  public void processToolkit(short ev) {
    if(ev == ToolkitConstants.EVENT_UNFORMATTED_SMS_PP_UPD){
      analyseAPDUSMS(); // using the stored reference
    }
  }
}
```

In a multithreaded context



```
try {
   // Code operating a sensitive process that will raise
   // an exception if deemed unsuccessful.
   ...
} catch (ExceptionType1 et1) {
   // The operation has failed in a specific way,
   // handle it accordingly.
   ...
} catch (ExceptionType2 et2) {
   // The operation has failed in another specific way,
   // handle it accordingly.
   ...
} finally {
   // Code executed whether an exception has been thrown
   // or not, caught or not.
   ...
}
```