Malicious Code on Java Card Smartcards: Attacks and Countermeasures

Wojciech Mostowski and Erik Poll
Digital Security
Radboud University Nijmegen

To be presented at CARDIS'2008

Overview

- Background and motivation
- Ways to *create* type confusion
  - experiments on actual cards
- Ways to *exploit* type confusion
  - experiments on actual cards
  - runtime countermeasures used
- Conclusions

Background

- Java Card smartcard allow multiple applets to be installed
  - installation strictly controlled by digital signatures
  - or completely disabled
    - eg on Dutch Java Card e-passport

- Most JavaCard smartcards have *no* bytecode verifier
  - could malicious, ill-typed applets do any damage?
  - not just to other applets, but also to platform
    - eg retrieving bytecode of platform implementation

- Java Cards do have a firewall
  - can this compensate for absence of bvc ?

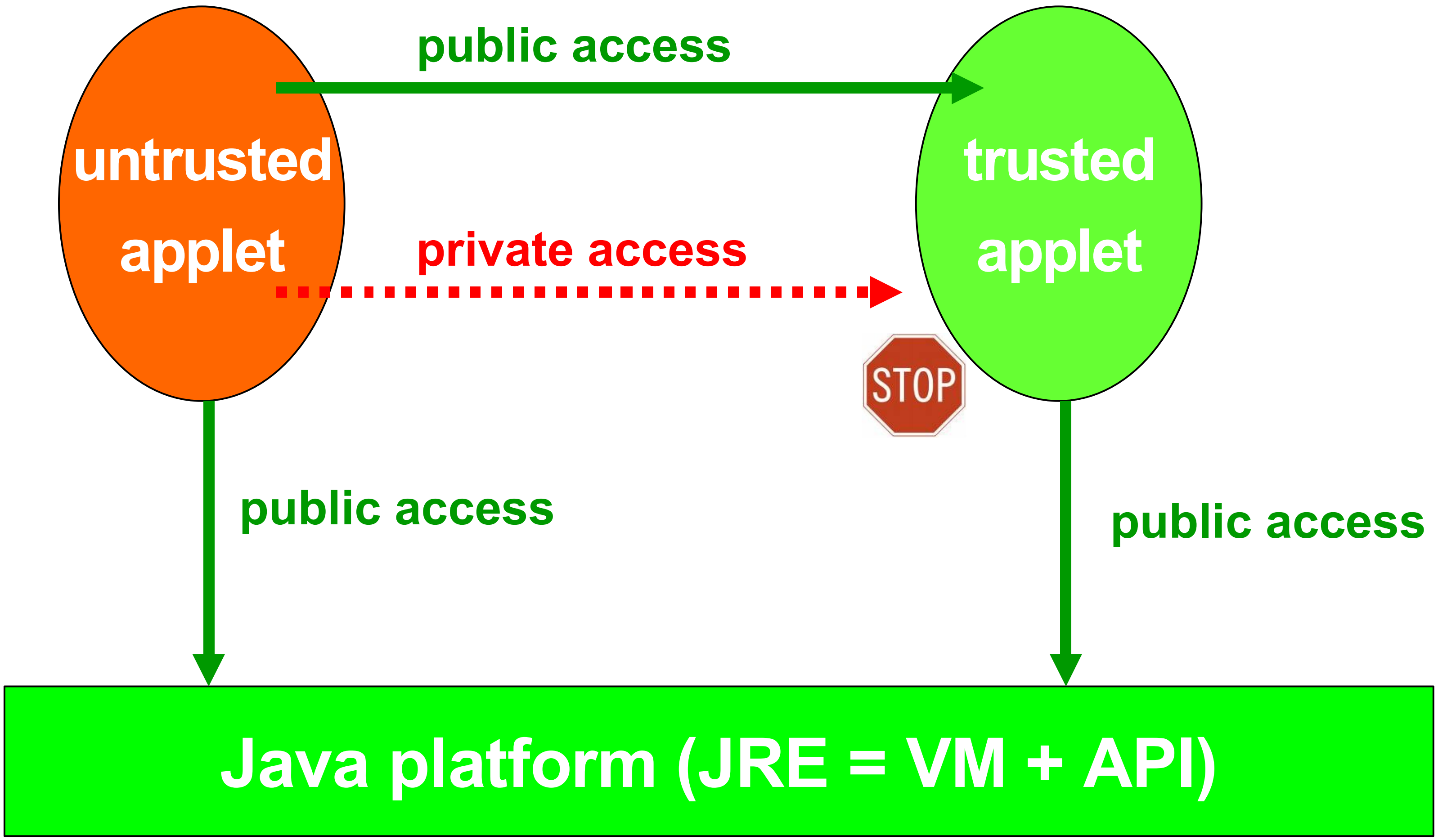
Two lines of defence on Java Card platform

- Type safety
  - enforced by bytecode verifier at *installation time*
  - optional; most cards use code signing instead
- Firewall
  - enforced by VM at *runtime*
  - restricts interactions between applets that type system allows
  - quite tricky!!

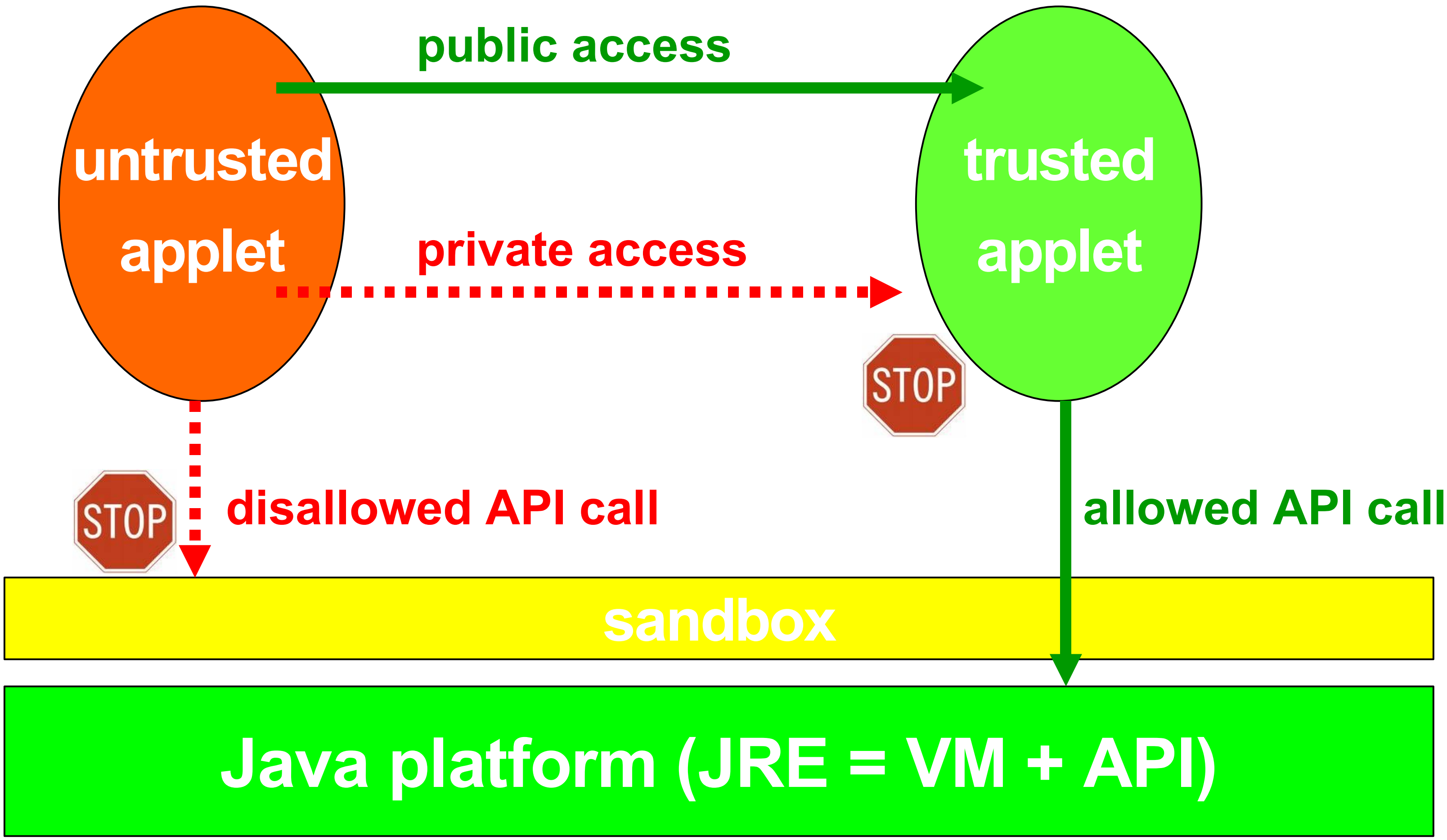
Are these defences *complementary* or *defense-in-depth*?

- what guarantees can firewall make about ill-typed code?

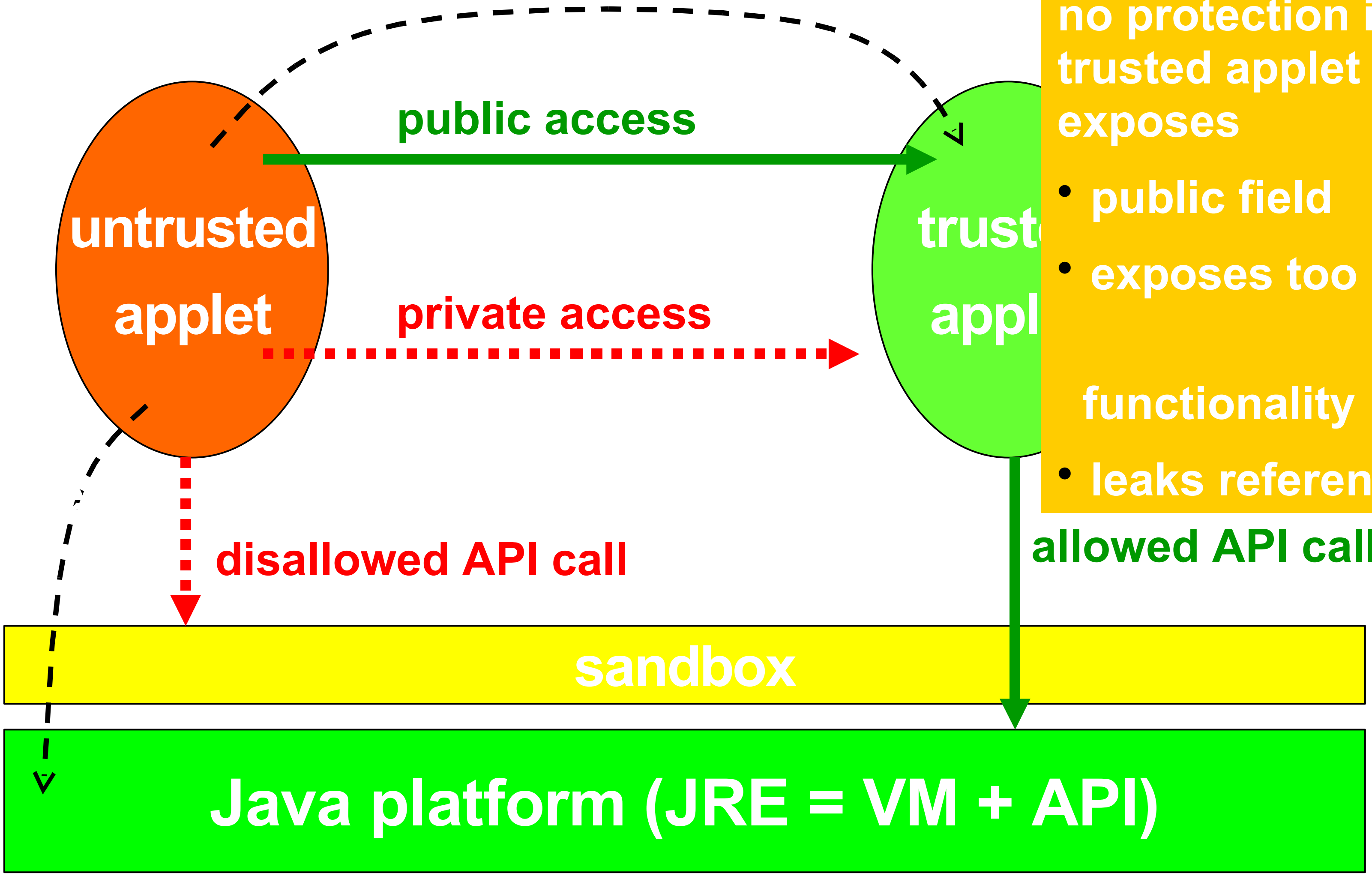
Java security: type-safety + visibility



Java security: type-safety + visibility + sandbox



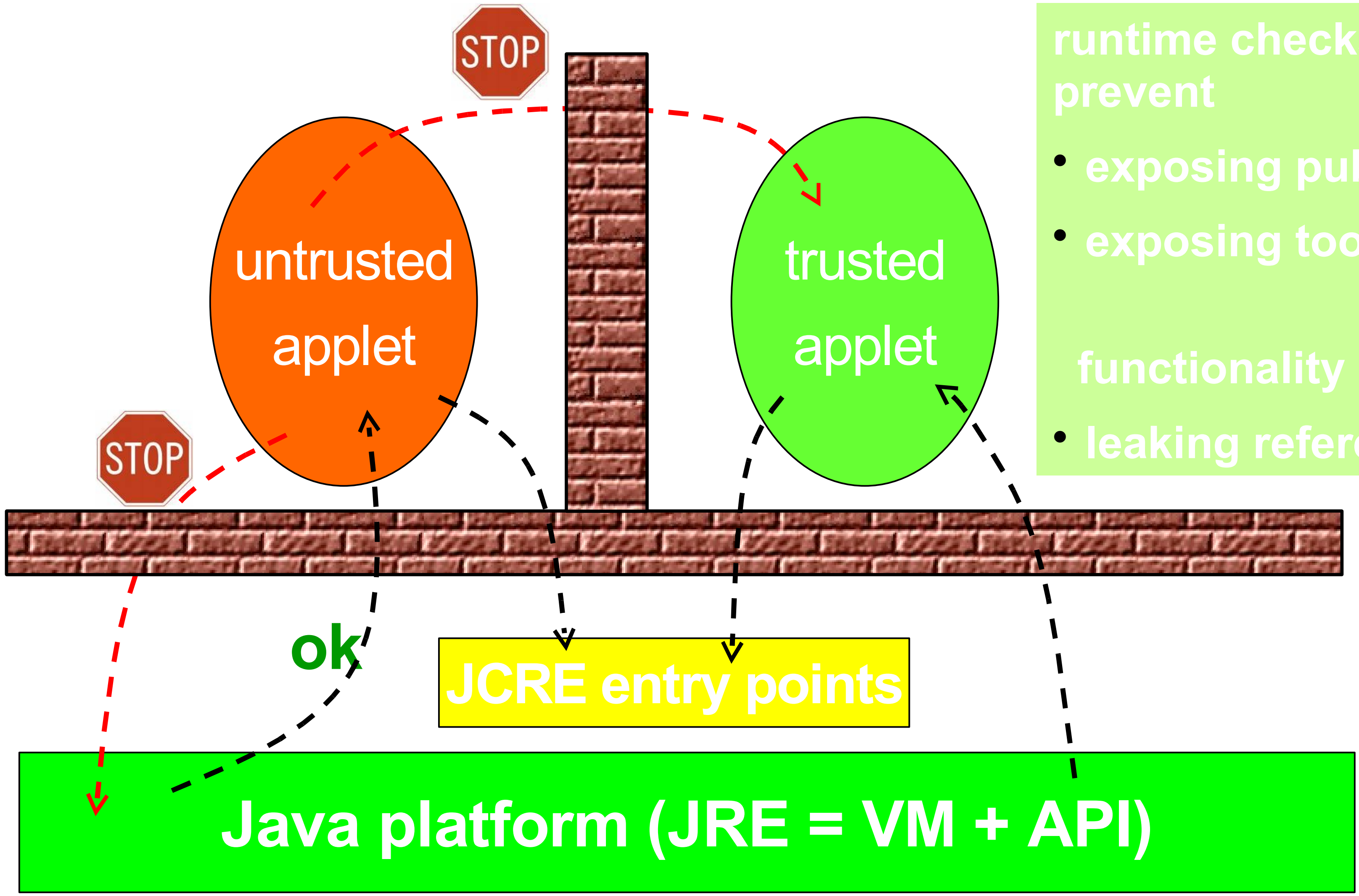
Java security: type-safety + visibility + sandbox



no protection if trusted applet or API exposes

- public field
- exposes too much functionality
- leaks reference

Java*Card* security: ... + *firewall*



runtime checks to prevent

- exposing public field
- exposing too much functionality
- leaking reference

Ill-typed code on Java Card

- NB Java Card specifications only define behaviour of well-typed programs
  - For ill-typed code, *all* bets are off....
    - This is case for VM spec, API specs, and JCRE specs
    - Eg a card could do a complete memory dump if a type error occurs. The specs allow this, but it's clearly unwanted.

- Only way to find out what happens:
  - test some cards

Rest of this talk

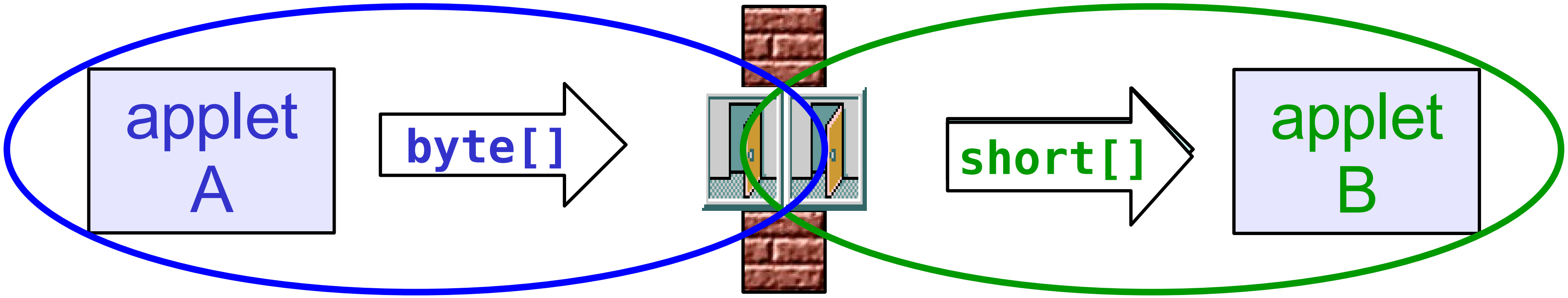
- Ways to *create* type confusion
  - how can be trick the VM in accessing the same piece of physical memory via references with different (incompatible) types?

- Ways to *exploit* type confusion to do some damage
  - ie. 'illegally' read or write memory in ways that should not be allowed

Way to *create* type confusion

- byte code editing
  - edit bytecode by hand to introduce type errors
    - or use some tool, eg by ST Microelectronics
- abusing shareable interface mechanism
  - two well-typed applets with type mismatch in shareable interface between them
- abusing transaction mechanism
  - exploring bug in transaction mechanism implementation
- fault injections?
  - introduce hardware fault (eg by laser) to corrupt memory that stores bytecode

Creating type errors with shareable interface



A thinks

```
void accessArray(byte[] a);
```

B thinks

```
void accessArray(short[] a);
```

Both applets type-correct (individually), compilable, and loadable.

Creating type errors using transactions

```
class MyApplet extend Applet {
 short[] s; // instance field
 byte[]  b; // instance field
 void someMethod(){
   short[] local = null;
   JCSystem.beginTransaction();
      s = new short[1]; s[0] = 24;
   JCSystem.endTransaction();
   ...
```

- s is either allocated *and* initialised, or neither, even if execution is interrupted by a card tear
- s reset to null if a card tear occurs during transaction

Creating type errors using transactions

```
class MyApplet extend Applet {
 short[] s; // instance field
 byte[]  b; // instance field
 void someMethod(){
   short[] local = null;
   JCSystem.beginTransaction();
      s = new short[1]; s[0] = 24;
      local = s;
   JCSystem.abortTransaction(); // resets s to null
   b = new byte[10];
   if ((Object)b ==(Object)local))...// true on some cards!!!
```

- buggy transaction mechanism reset only `s` to null, not `local`

One role of formal methods

- (Too) hard to formalise

  $\Longrightarrow$

  Hard to implement

  $\Longrightarrow$

  Security problems are not unlikely....

  - **For example, the transaction mechanism is very tricky when allocating objects inside transactions**
    - **see Nicolas Rousset's thesis, Chapter 3**

Experiments creating type confusion

	A211	A221	B211	B22	B221	C211	C211'	D211
on-card bcv?	✓	✓	✓	✓	✓	yes -	yes -	✓
bytecode editing	✓	✓	✓	✓	✓	-	-	-
abusing shareable	-	-	✓	-	✓	✓	-	✓
abusing transaction								

- ill-typed code possible on card C211 *despite* bcv!!
  - because of buggy transaction mechanism
- cards with bcv don't allow shareable interfaces
  - and hence are not standard-compliant?

Ways to exploit with type confusion

- confusing byte arrays and short arrays
  - possibly accessing twice as much memory
- accessing array as object
  - possibly set the length field
- accessing object as array
  - possibly doing pointer arithmetic (using numeric value as references)
- confusing objects of various types
  - possibly accessing outside memory or doing pointer arithmetic

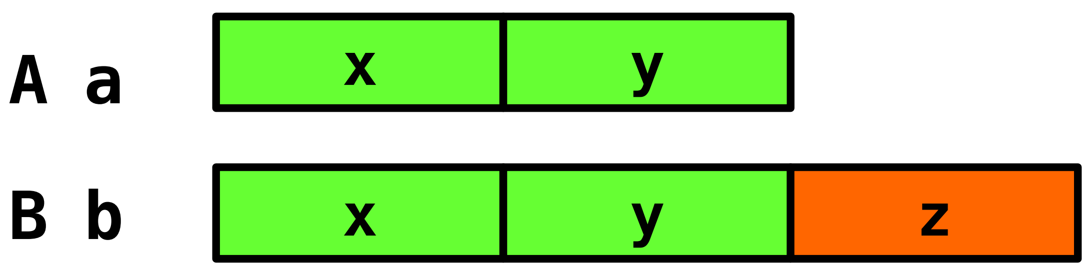
Confusing object types

```
class A {
     final short x;

     short y;
}
```

```
class B {
     short x,
     Object y, z;
}
```

What if VM can be tricked to access object `A a` as if it is of type `B`?



We might be able to

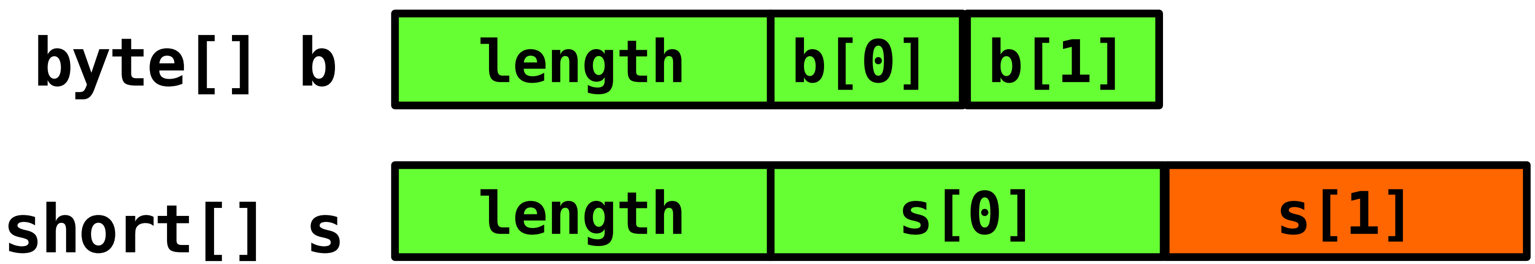
- access memory outside bounds (namely a.z)
- do pointer arithmetic (using a.y)
- modify final fields (namely a.x)

Accessing byte array as short array

```
byte[] b = { 23, 24}; // b.length = 2
```

If we acccess `byte[] b` as `short[] s`, then

- what is s.length ?
- what is s[1]?



If VM can be tricked in treating `byte[]` as `short[]`,

physical array size might double,

- allowing access outside array bounds

Accessing object as array (1) [M Witteman, RSA2003]

```
public class FakeArray {
    short length = 0x7FFF;
    short  x =  23 ;
}
```

FakeArray a	0x7FF	23		
short[] s	length	s[0]	s[1]	s[2]

If VM can be tricked in treating `FakeArray` as `short[]`,

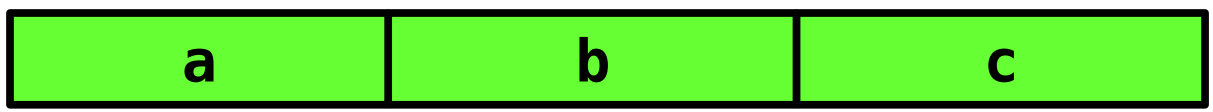
maybe array lengths can be set

- accessing memory way outside the object's bounds
- depending on layout of objects and arrays in memory

Accessing object as array (2)

```
public class MyObject {
      A a = new A();
      B b = new B();
      C c = new C();
}
```

`MyObject o`



Treating `MyObject o` as a `short[] s`, what happens with

- `s[0] = s[1];` ?
  - swapping references like this works on some cards
- `s[0] = 24612;` ?
  - spoofing a reference like this fails on nearly all cards

Runtime defense mechanisms

Some cards employ runtime countermeasures:

- **Physical Bounds Checking (PBC)**

  array bounds are checked using physical sizes rather than logical sizes

  - confusing `byte[]` and `short[]` becomes harmless

- **Object Bounds Checking (OBC)**

  object bounds checked at runtime just like array bounds

  - confusing objects and arrays becomes less harmfull ;
    - no access beyond object's original size

- **Runtime Type Checking (RTC)**

  object types are checked at runtime for every VM step

  - all attempts at type confusion become harmless

Experiments running ill-typed code

	A2 11 PB C	A2 21 O B C	B2 11 R T Є	B2 2 O B C	B2 21 R T Є	C2 11 bc v	C21 bcv 1	D2 11
protection?								
byte-short[ ]	✓	✓		-		⚡	-	⚡
object as array	-	✓				⚡	-	⚡
array as object	⚡	✓		✓		⚡	-	⚡
reference switching	-	✓	-	✓	-	-	-	nt
~ in AIDs	-	⚡	-	⚡	-	-	-	nt
reference spoofing	-	-	-	-	-	-	-	⚡

✓ ⚡

Reference switching in AID objects

```
package javacard.framework
public class AID {
    final byte[] theAID;
    ...
}
```

- reference switching on some cards allows `theAID` field in `AID`s (Applet IDentifiers) to be changed to point to other byte arrays
  - this allows system-owned `AID`s to be changed
  - `AID`s are used for identifying applets on the card...

Conclusions

- Many attacks, some with harmful results
- On-card bcv not sufficient
  - if there are bugs in transaction mechanism...

  Also, on-card bcv limits functionality:

    no Shareable Interfaces between applets
- (Increasingly?) cards employ runtime countermeasures
  - runtime checks more robust that static checks!
  - runtime typechecking is best countermeasure
    - downside: performance overhead?
- All this applies *only* to open cards
  - no threat on most (all?) Java Cards in the field