

# Java Card

- Smartcards
- Java Card
- Demos



# Smart Cards

# Smartcards

**Credit-card size piece of plastic with embedded chip, for storing & processing data**

## **Standard applications**

- **bank cards**
- **SIM's in GSM mobile phones**
- **pay-TV**
- **ov-kaart**
- **...**

# Smartcards

A smartcard is a **miniature computer** with

- **limited resources**
  - **ROM: 16-64K** : OS + programmatuur
  - **EEPROM: 4-64K**: persistent data storage
  - **RAM: 256 byte-4K**: scratch-pad memory
- **minimal I/O**
  - **just a serial port** no keyboard, screen, mouse, ...

**External power supply & clock.**



# Smart vs stupid cards

**'Smart'** refers to **ability to compute**.

**'Stupid'** cards, providing only provide memory, and no processing power, namely

- **magnetic stripe cards**
- **memory-only chip cards**

# Why use smartcards ?

Smartcard can **securely** store and process information:  
**data and program stored on smartcard cannot (easily) be read, copied, or alerted.**

***NB unlike a PC!***

**Typical application:**

- **storing sensitive information:**  
eg. cryptographic keys, passwords, pincodes & pin counter, chipknip balance.
- **using/changing this information:**  
eg. de/encrypt, check pincode, change balance
- whilst **protecting against unauthorized use.**

# Why use smartcards ?

Typical use of smartcard for **authentication**:

- X sends a random challenge  $c$  to smartcard
- smartcard responds with  $f_K(c)$ , ie.  $c$  encrypted with some secret key  $K$ .
- X checks this result

Eg. used in GSM – for network to authenticate phone –, and in chipknip – for point-of-sale to authenticate chipknip and for chipknip to authenticate oplaadpunt.

**NB**

- **secret key never leaves the smartcard!**
- **observing traffic between smartcard and terminal leaks no useful information, so communication with smartcard can be over untrusted network.**

## Why use smartcards ?

Smartcards typically part of the **Trusted Computing Base (TCB)**, ie. **the part of the infrastructure that we *have to trust***.

You want your TCB to be as reliable – and hence as small & simple – as possible!

*Eg. for a telecom network operator, the GSM SIM is part of the TCB, but rest of the mobile phone is not.*

# Smartcard limitations

**Primitive I/O is a limitation:**

***eg. chipknip user has to trust the terminal, for deducting the right amount and to keep pincode secret***

**Smart card with displays, keyboards, fingerprint detectors, etc. are being developed ...**

**Contactless smartcards rapidly getting better.**

# Security of smartcards

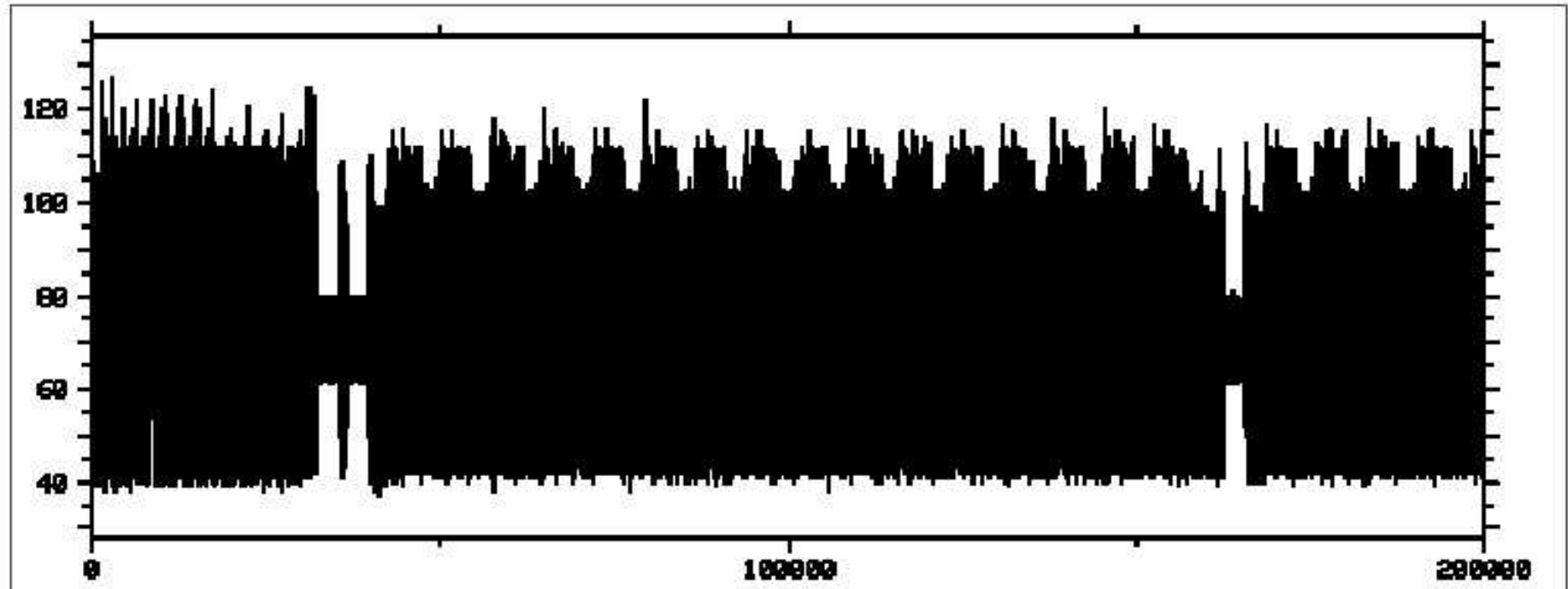
Smartcard is **tamper-resistant** and **tamper-evident**, to a degree, but not **tamper-proof**.

Smart card can be attacked by for instance

- **differential power analysis (DPA)**, analysis of power consumption of card
- **fault injection**, eg. temporarily dip power supply, flash light on chip
- **chemical etching/staining** to read memory contents
- **probing** to analyse traffic on bus
- **focused ion beam** to remove/deposit material on chip surface

*TNO-EIB in Delft are experts at this.*

# Attacking smartcards: DPA



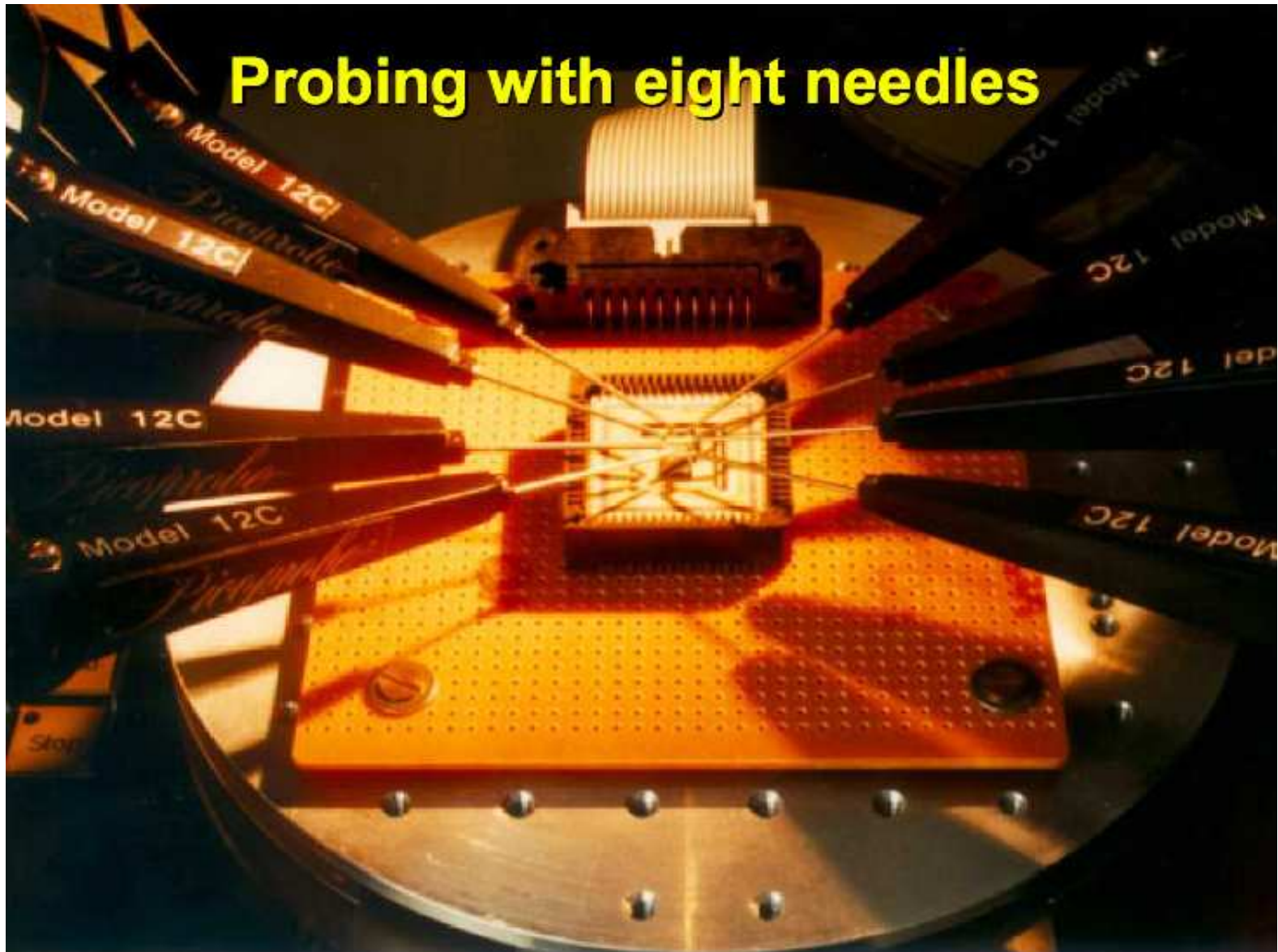
**Power analysis of smart card doing a DES encryption**

# Attacking smartcards: Probing



**Sub micron probe station**

# Attacking smartcards: Probing



# Attacking smartcards: Staining



# Old vs new smartcards

## Traditional smartcard

- programmed in machine-code, specific to the chip
- one application, burnt into ROM

## Newer smartcards

- application written in high-level language
- compiled into byte code
- stored in EEPROM
- interpreted on card

# Old vs new smartcards

## Advantages of newer smartcards

- cheaper development
- quicker time-to-market
- vendor-independence

and

- **multi-application**: several applications can be installed on one smartcard
- **post-issuance download**: applications can be added, updated, removed on card already issued to customer (cf. web browser)

# Java Card

# Java Card

Dialect of Java for programming smartcards

Slimmed down **due of hardware constraints**

- not 32-bit but 16-bit arithmetic
- no doubles, no floats, no strings, no threads, ...
- no / optional garbage collection
- very limited API

plus special features **due to hardware peculiarities**

- persistent (EEPROM) vs transient (RAM) memory
- applet firewall
- transaction mechanism
- smartcard I/O with APDUs using ISO7816

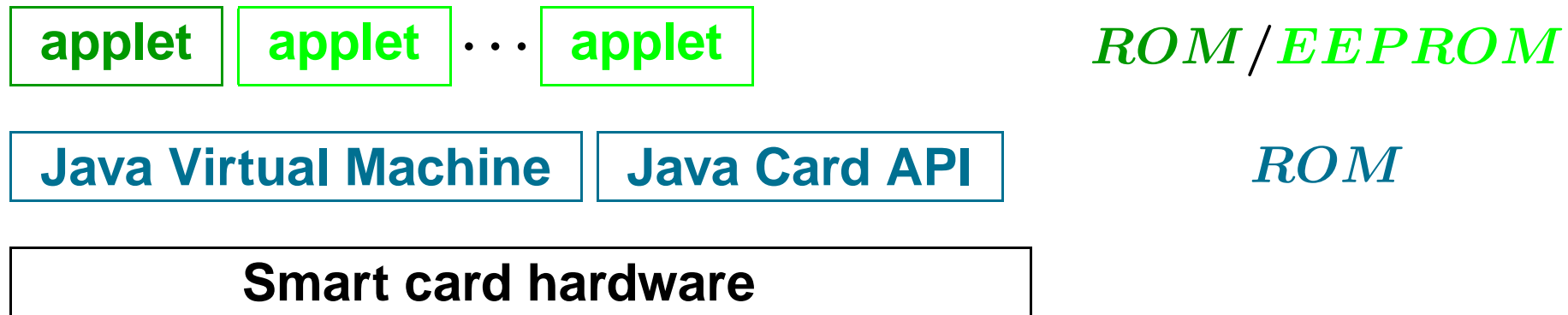
# Java Card platform

The Java Card platform consists of

- **Virtual Machine (VM)**  
for interpreting bytecodes
- **API**, providing
  - basic classes (eg. applet and PIN)
  - “interface” to OS (eg. APDU class for I/O)

Partly written in Java Card, partly **native code**

# JavaCard architecture



# J2ME

**Java 2 Micro Edition (J2ME)** is the bigger brother of Java Card.

J2ME has a **bigger memory footprint**, as is intended for use in **mobile phones, PDAs, set-top boxes, etc.**

Eg MIDP applets that can execute on a mobile phone

**Lots of dedicated APIs** for TV, Phone, BlueTooth, ...

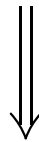
# Java Card & smartcard hardware

- **ROM:**  
for program code of VM, API, and pre-installed applets
- **EEPROM** for persistent data:  
for objects and their fields (ie. heap) and code of downloaded applets
- **RAM** for transient data:  
for stack and specially allocated scratchpad arrays

Writing to EEPROM is slower & consumes more power than writing to RAM, and EEPROM lifetime limited to certain number of writes.

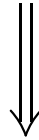
# Producing & installing applets

Java (Card) source code **A.java**



**java compiler**

Java (Card) byte code **A.class**



**cap converter**

'compressed' Java Card cap file **A.cap**



**downloading**



# Bytecode verification

Bytecode verification, ensuring **type-safety** of code, is **CRUCIAL** for Java (Card) security!

Due to hardware constraints, bytecode verification on smartcard is difficult/not possible.

Instead, **code signing** is used: **bytecode verification done off-card, and only digitally signed code is trusted by the platform.**

# Java(Card) security by typing (1)

**Access-control**, eg. no access to **private fields** of other objects

```
PIN pin;  
  
...  
for (i=0; i++; i<4)  
    System.out.println(pin.thePin[i]);  
  
...  
pin.tryCounter = 9999;
```

In fact, Java Card provides a firewall between applets, which forbids most interaction between applets.

## Java(Card) security by typing (2)

**Type-safety**, eg. **no forging of pointers** (as in C doing pointer arithmetic with `&`) or **accessing outside array bounds**

...

```
PIN pin;
```

```
byte[] a = new byte[1];
```

...

```
a = pin; // type error !!
```

```
for (i=134; i++; i< 137)
```

```
    System.out.println(a[i]);
```

...

```
a[36] = 9999; // access outside array bounds !
```

# Java Card API

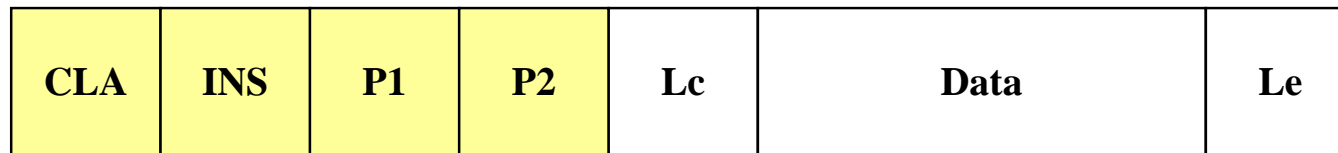
Java Card peculiarities for which the API provides support include

- a **APDU** class for communication of smartcard with terminal
- a **transaction mechanism** to cope with **card tears**, ie. interruptions to power supply.

# APDU

(Low-level!) communication with smartcard via **byte array buffer aka APDU**, following **ISO7816**

- Terminal sends card a **command APDU**



- **CLA: Class byte**
  - **INS: Instruction byte**
  - **P1, P2: Parameters**
  - **Lc: Length data block**
  - **Le: Expected length response**
- Card sends terminal a **response APDU**



- **SW1, SW2: Status words**

# Transactions

Sudden loss of power possible at any time due to **card tear**, ie. pulling smartcard from reader.

**Transaction mechanism** provides atomic updates in event of card tears (like transactions in databases).

*For example, in*

...

```
JCSystem.beginTransaction( );
```

```
x++ ;
```

```
y++ ;
```

```
JCSystem.endTransaction( );
```

*either both x and y are increased, or neither is.*

# RMI

**APDU format very clumsy & antiquated.**

**Java Card 2.2 supports **Remote Method Invocation (RMI)****

**RMI means one VM (the JVM on the terminal) invokes method on another VM (the JCVM on the smartcard).**

**For Java Card, such a method invocation (incl. its parameters and any return value it produces) is translated into APDUs handled by the platform.**