BUSLab

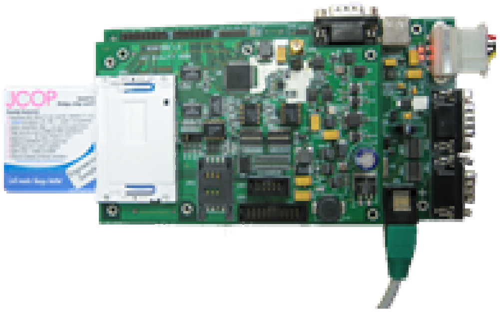
Brno University Security Laboratory

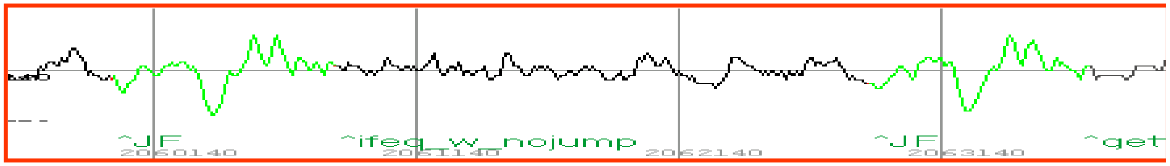
Cryptographic smart cards & Java Card & PKI tutorial

Jiří Kůr, Tobiáš Smolka, Petr Švenda
Masaryk University, Czech Republic, Brno
{xkur,xsmolka,svenda}@fi.muni.cz

What’s in pipeline?

- Cryptographic smart cards
  - Basic details and specifications
- Applications
  - Common applications
  - Custom build systems
- Programming in Java Card
  - PC and card side
- Smart card in existing applications
- (Attacks)





What to do during the introduction ☺

Big thanks to Tobiáš Smolka!

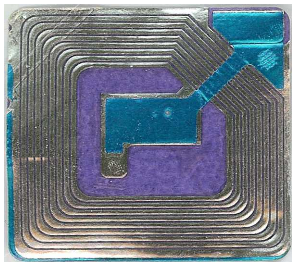
- Install from DVD
  - VirtualBox to host system
  - smart card reader drivers into host system
  - insert smart card into reader
- Run Ubuntu image
  - user: europen, password: europen
- Connect USB reader into image (Devices→USB devices)
- Run terminal: pcsc_scan
- Run NetBeans
- Rebuild selected project (e.g., JOpenPGPCard)
- Upload applet to smart card (Run→Test project)

Some troubleshooting

- Wait sufficiently before actions (sleep 5)
- Is **`lsusb`** and **`pscs_scan`** detecting reader?
- Try to abort and restart program
- Try to remove and insert again card
- Try to remove and add USB from physical slot
- Try to remove and add USB device in VirtualBox
- Try to disable and enable USB reader in Ubuntu
- Try to restart virtual machine
- Note:
  - PCMCIA readers cannot be propagated into VirtualBox
  - Missing driver for Smart Card on Windows is NOT problem

Smart card basics

Basic types of (smart) cards

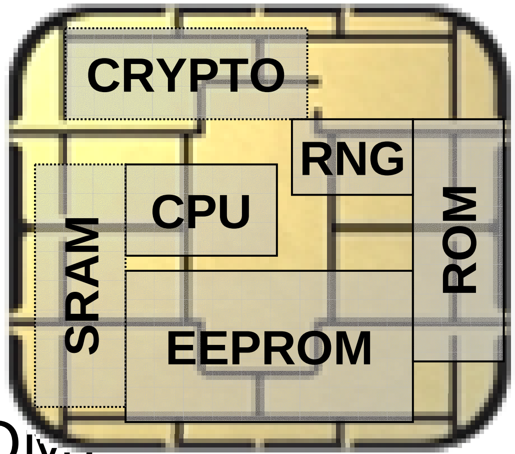


- Contactless “barcode”
  - Fixed identification string (RFID, < 5 cents)
- Simple memory cards (magnetic stripe, RFID)
  - Small write memory (< 1KB) for data, (~10 cents)
- Memory cards with PIN protection
  - Memory (< 5KB), simple protection logic (<$1)

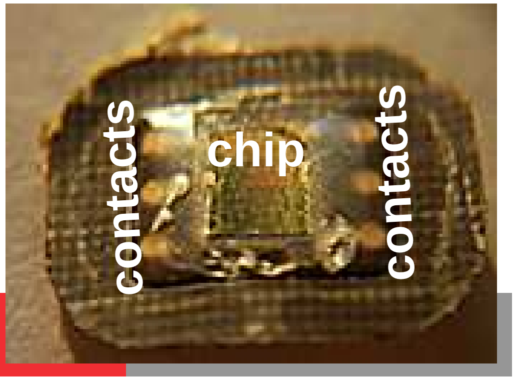
Basic types of (smart) cards (2)

- Cryptographic smart cards
  - Support for (real) cryptographic algorithms
  - Mifare Classic ($1), Mifare DESFire ($3)
- User-programmable smart cards
  - Java cards, .NET cards, MULTOS cards ($10-$30)

Cryptographic smart cards



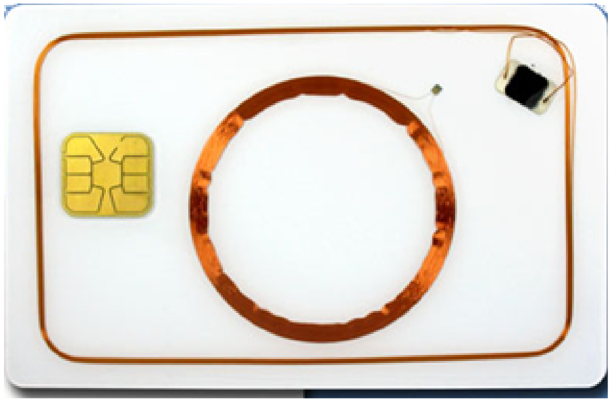
- SC is quite powerful device
  - 8-32 bit procesors @ 5-20MHz
  - persistent memory 32-100kB (EEPROM)
  - volatile fast RAM, usually <<10kB
  - truly random generator
  - cryptographic coprocessor (3DES, RSA-2048...)
- 5.5 billion units shipped in 2010 (EUROSMART)
  - 4.2 billion in Telcom, 880Mu payment and loyalty
  - 370Mu contactless smart cards

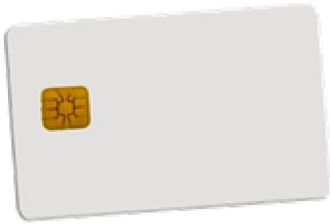


Smart card is programmable

- Programmable (C, Java Card, .NET…)
  - (Java) Virtual Machine
  - multiple CPU ticks per bytecode instruction
- interfaces
  - I/O data line, voltage and GND line (no internal power source)
  - clock line, reset lines

Smart cards forms





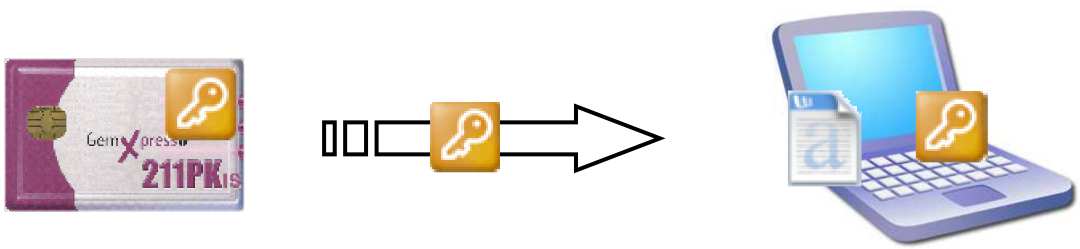
- Many possible forms
  - ISO 7816 standard
  - SIM size, USB dongles, Java rings…
- Contact(-less), hybrid/dual interface
  - contact physical interface
  - contact-less interface
    - chip powered by current induced on antenna by reader
    - reader->chip communication - relatively easy
    - chip->reader – dedicated circuits are charged, more power consumed, fluctuation detected by reader
  - hybrid card – separate logics on single card
  - dual interface – same chip accessible contact & c-less

Main advantages of crypto smart cards

- High-level of security
- Fast cryptographic coprocessor
- Programmable secure execution environment
- Secure memory and storage
- On-card asymmetric key generation
- High-quality and very fast RNG
- Secure remote card control

Smart card as a secure carrier

- Key stored on the card, loaded to the PC before encryption/signing, then erased
- High speed encryption (>>MB/sec)
- Attacker with access to the PC during encryption will obtain the key
  - key protected for transport, but not during usage



Example: Secret file(s) inside PKCS#11 Security Token used by TrueCrypt

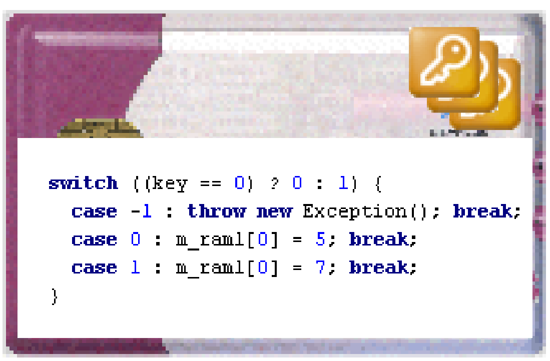
Smart card as an encryption/signing device

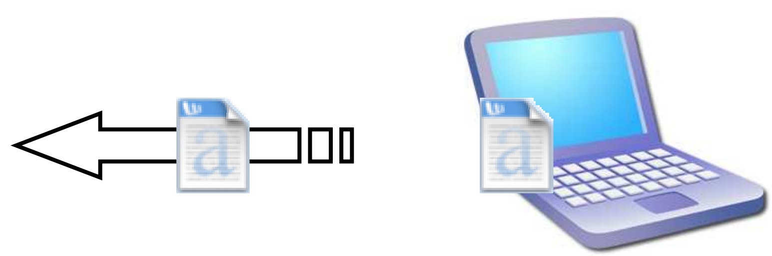
- PC just sends data for encryption/signing
- Key never leaves the card
  - protected during transport and usage
- Attacker must attack the smart card
  - or wait until card is inserted and PIN entered!
- Low speed encryption (~kB/sec)
  - mainly due to the communication speed

Example: Private signature key inside OpenPGP card used by GPG

Smartcard as computational device

- PC just sends input for application on smart card
- Application code & keys never leave the card
  - smart card can do complicated programmable actions
  - can open secure channels to other entity
    - secure server, trusted time service…
    - PC act as a transparent relay only (no access to data)
- Attacker must attack the smart card





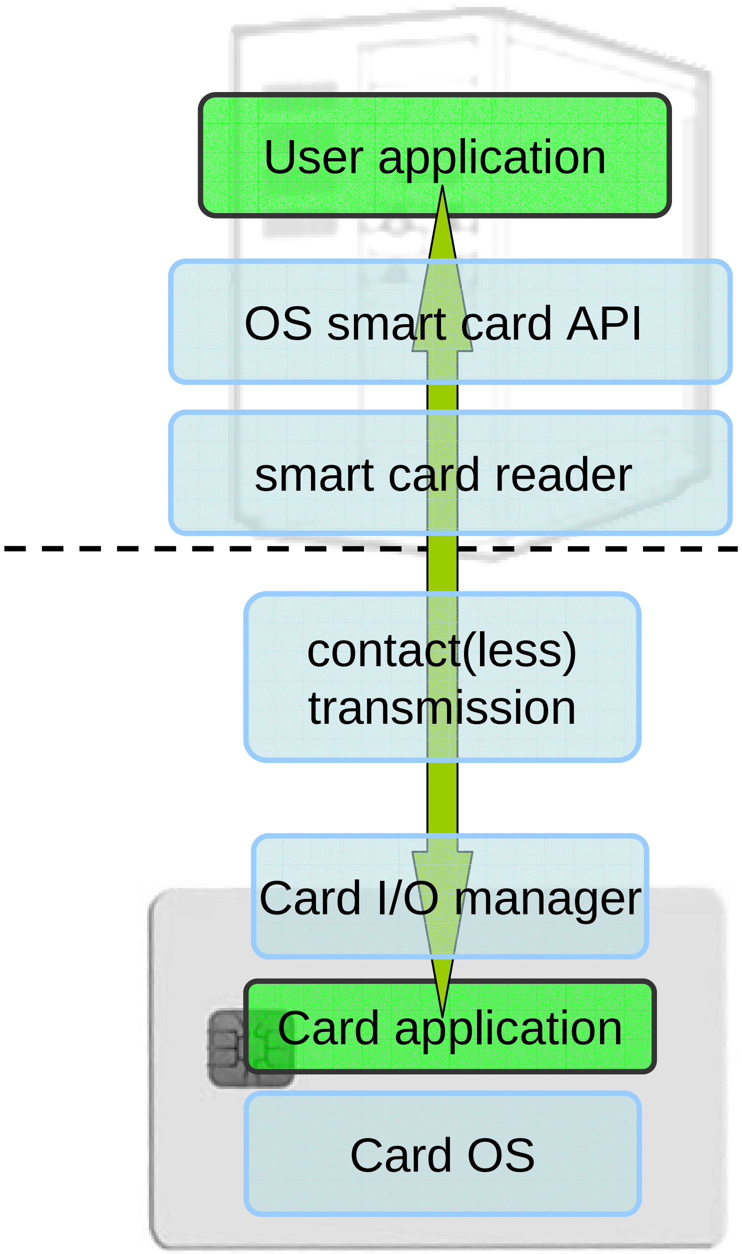
Example: Satellite Pay TV cards

Smart cards are used for…

- GSM SIM modules
- Bank payment card (EMV standard)
- Digital signatures (private key protection)
- System authentication
- Operations authorizations (PKI)
- ePassports
- Multimedia distribution (DRM)
- Secure storage and encryption device
- …

Main standards

- ISO7816
  - card physical properties
  - physical layer communication protocol
  - packet format (APDU)
- PC/SC, PKCS#11
  - standardized interface on host side
  - card can be proprietary
- MultOS
  - multi-languages programming, native compilation
  - high security certifications, often bank cards
- Java Card
  - open programming platform from Sun
  - applets portable between cards
- Microsoft .NET for smartcards
  - similar to Java Card, relatively new
  - applications portable between cards
- GlobalPlatform
  - remote card management interface
  - secure installation of applications



Supported algorithms

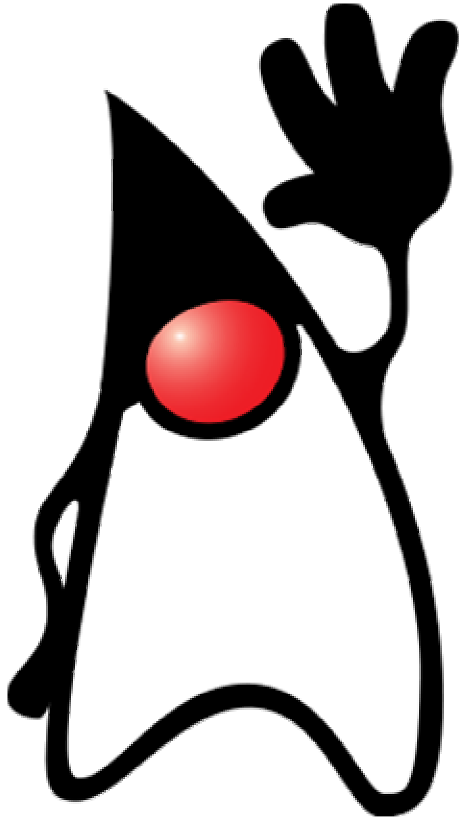
- Symmetric cryptography
  - DES, 3DES, AES (~10kB/sec)
- Asymmetric cryptography
  - RSA 512-2048bits, 2048 often only with CRT
  - Diffie-Hellman key exchange, Elliptic curves
    - rarely, e.g., NXP JCOP 4.1
  - on-card asymmetric key generation
    - private key never leaves card!
- Random number generation
  - hardware generators based on sampling thermal noise…
  - very good and fast (w.r.t. standard PC)
- Message digest
  - MD5, SHA-1, (SHA-2)
- *http://www.fi.muni.cz/~xsvenda/jcsupport.html* for more

Our card: Gemalto TOP IM GX4

http://www.gemalto.com/dwnld/5304_TOP_GX4_May10.pdf

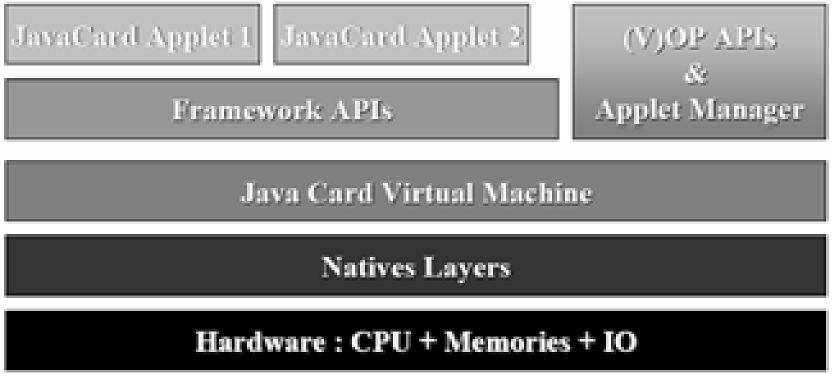
- Java Card 2.2.1, Global Platform 2.1
- 72k EEPROM
- 3DES, AES (128, 192, 256)
- RSA up to 2048bit
- (MD5), SHA-1
- TRNG
- Contact interface: T=0, T=1
- FIPS 140 and CC EAL 4+ certifications
- Garbage collection
- *[Mifare 1k is separate chip embedded in plastic]*

Java Card basics



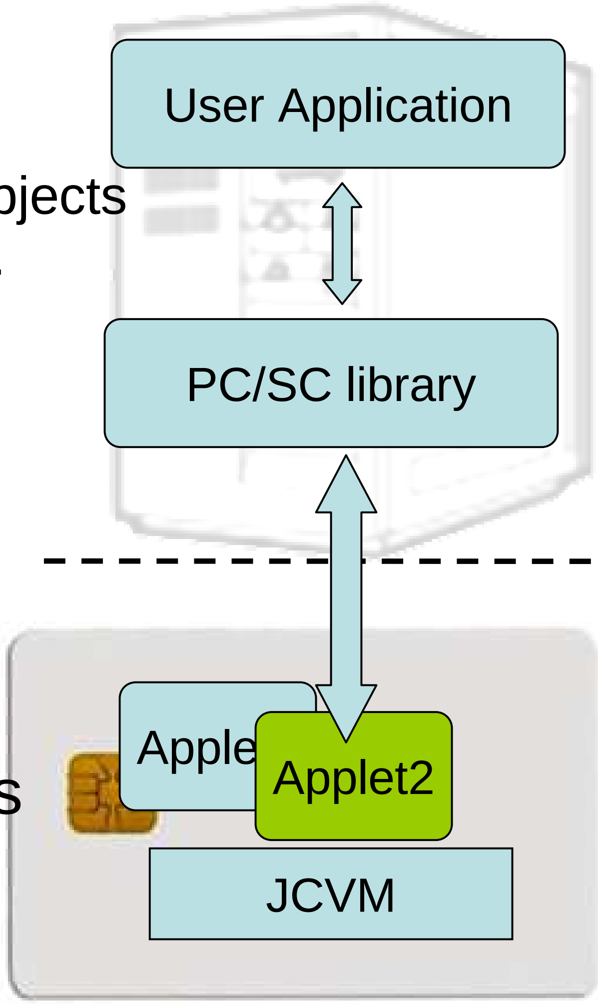
Java Card

- Maintained by Sun Microsystems (Oracle)
- Cross-platform and cross-vendor applet interoperability
- Freely available specifications and development kits
  - *www.oracle.com/technetwork/java/javacard/index.html*
- Java Card applet is Java-like application
  - uploaded to a smart card
  - executed by the Java Card Virtual Machine
  - installed once (“running” until deleted)
  - suspended on power loss
  - data preserved after power loss
  - code restarted on power up



Java Card applets

- Writing in restricted Java syntax
  - byte/short (int) only, missing most of Java objects
- Compiled using standard Java compiler
- Converted using Java Card converter
  - check bytecode for restrictions
  - can be signed, encrypted…
- Uploaded and installed into smartcard
  - executed in JC Virtual Machine
- Communication using APDU commands
  - small packets with header



Java Card versions

- Java Card 2.1.x/2.2.x
  - widely supported versions
  - basic symmetric and asymmetric cryptography algorithms
  - PIN, hash functions, random number generation
  - transactions, utility functions
- Java Card 2.2.2
  - last version from 2.x series
  - significantly extended support for algorithms and new concepts
    - long “extended” APDUs, BigNumber support
    - biometric capability
    - external memory usage, fast array manipulation methods…
- Java Card 3.x
  - classic and connected editions (see slides for more info)

Version support

- Need to know version support for your card
  - convertor adds version identification to package
  - unsupported version will fail during card upload
  - (use Converter from JC SDK 2.2.1)
- Available cards supports mostly 2.x specification
  - rest of presentation will focus on 2.x versions
- Our card (Gemalto TOP IM GX4) is 2.2.1

Java Card 2.x *not* supporting

- Dynamic class loading
- Security manager
- Threads and synchronization
- Object cloning, finalization,
- Large primitive data types
  - float, double, long and char
  - usually not even int (4 bytes) data type
- Most of std. classes
  - most of java.lang, Object and Throwable in limited form
- Garbage collection
  - some card now do but slow and unreliable

Java Card 2.x supports

- Standard benefits of the Java language
  - data encapsulation, safe memory management, packages, etc.
- Applet isolation based on the Java Card firewall
  - applets cannot directly communicate with each other
  - special interface (Shareable) for cross applets interaction
- Atomic operations using transaction mode
- Transient data
  - fast and automatically cleared
- A rich cryptography API
  - accelerated by cryptographic co-processor
- Secure (remote) communication with the terminal
  - if GlobalPlatform compliant (secure messaging, security domains)

Java Card 3.x

- Recent major release of Java Card specification
  - significant changes in development logic
  - two separate branches – Classic and Connected edition
- Java Card Classic Edition
  - legacy version, extended JC 2.x
  - APDU-oriented communication
- Java Card Connected Edition
  - smart card perceived as web server (Servlet API)
  - TCP/IP network capability, HTTP(s), TLS
  - supports Java 6 language features (generics, annotations…)
  - move towards more powerful target devices
  - focused on different segment then classic smart cards

Exercise / developing simple applet

Simple applet - requirements

1. Write Java Card applet
   - able to receive data, change it and send back
   - e.g., add 1 to every input byte
2. Install applet on smart card
3. Write simple Java communication program
   - send data to Java Card applet

Necessary tools

- Several tool chains available
  - both commercial (RADIII, JCOPTools, G&D JCS Suite)
  - and free (Sun JC SDK, Eclipse JC plugin…)
- We will use:
  - NetBeans 6.8 or later
  - Java Standard Edition Development Kit 1.3 or later
  - Apache Ant 1.7 or later, GPShell 1.4.2
  - Java Card Development Kit 2.1.2
  - Java Card Ant Tasks (from JC SDK 2.2.2)
- Everything already preinstalled in Ubuntu image

Caution – pre-configured project!

- We will use already pre-configured project
  - see your DVD
- VirtualBox Ubuntu image
  - NetBeans & all SDKs already installed
  - build.xml modified to include Ant tasks
  - project.properties contains correct paths
  - upload script prepared for target card
  - pcsclite, opensc…
- Compilation details at http://www.0x9000.org/

JC development process

1. Subclass javacard.framework.Applet

2. Compile Java→*.class (Java 1.3 binary format)

3. Convert *.class→*.jar/cap (Java Card Convertor)

4. Upload *.jar/cap →smart card (GPShell)

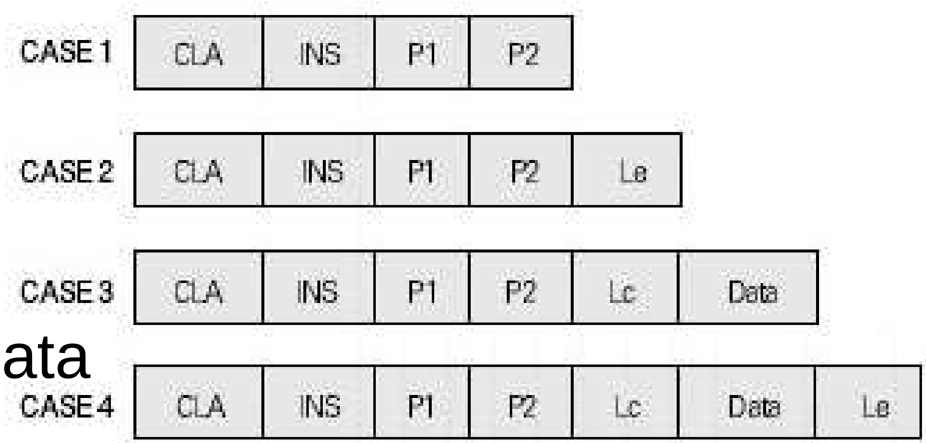
5. Install applet (GPShell)

6. Write user Java app (javax.smartcardio.*)

7. Use applet on smart card (APDU)

APDU (Application Protocol Data Unit)

- APDU is basic logical communication datagram
  - header (5 bytes) and up to ~256 bytes of user data
- Header format
  - CLA – instruction class
  - INS – instruction number
  - P1, P2 – optional data
  - Lc – length of incoming data
  - Data – user data
  - Le – length of the expected output data



Response APDU (R-APDU)

- Response data + status word (2 bytes)
  - 0x9000 - SW_NO_ERROR, OK
  - 0x61** - SW_BYTES_REMAINING_**
  - see javacard.framework.ISO7816 interface
  - other status possible (GlobalPlatform, user defined)
- May require special command to read out
  - first response is just status word (0x61**)
  - *00 C0 00 00* ** or *C0 C0 00 00* ** APDU
    - ** is number of bytes to read out

Simple Java Card applet - code

1. Develop Java Card Applet (NetBeans)
    a. subclass `javacard.framework.Applet`
    b. allocate all necessary resources in constructor
    c. select suitable CLA and INS for your method
    d. parse incoming APDU in `Applet::process()` method
    e. call your method when your CLA and INS are set
    f. get incoming data from APDU object (`getBuffer()`, `setIncomingAndReceive()`)
    g. use/modify data
    h. send response (`setOutgoingAndSend()`)

```
package example;
import javacard.framework.*;

public class HelloWorld extends Applet {
   protected HelloWorld()  {
      register();
   }
   public static void install(byte[] bArray, short bOffset, byte bLength) {
      new HelloWorld();
   }
   public boolean select() {
      return true;
   }
   public void process(APDU apdu) {
      // get the APDU buffer
      byte[] apduBuffer = apdu.getBuffer();
      // ignore the applet select command dispached to the process
      if (selectingApplet()) return;
      // APDU instruction parser
      if (apduBuffer[ISO7816.OFFSET_CLA] == CLA_MYCLASS) &&
         apduBuffer[ISO7816.OFFSET_INS] == INS_MYINS)) {
         MyMethod(apdu);
      }
      else ISOException.throwIt( ISO7816.SW_INS_NOT_SUPPORTED);
   }
   public void MyMethod(APDU apdu) { /* ... */ }
}
```

include packages from javacard.*

extends Applet

Called only once, do all allocations&init HERE

Called repeatedly on application select, do all temporaries preparation HERE

Called repeatedly for every incoming APDU, parse and call your code HERE

JavaCard communication cycle

1. (Applet is already installed)
2. Reset card (plug smart card in, software reset)
3. Send SELECT command (00 0a 04 00 xxx)
   - received by Card Manager application
   - sets our applet active, select() method is always called
4. Send any APDU command (of your choice)
   - received by process() method
5. Process incoming data on card, prepare outgoing
   - encryption, signature…
6. Receive any outgoing data
   - additional special readout APDU might be required
7. Repeat again from step 4
8. (Send DESELECT command)
   - deselect() method might be called

Simple Java Card applet – compile&convert

1. Compile with standard Java Compiler (javac)
   - Java source/binary format version 1.3
   - libraries from Java Card SDK (api.jar)
2. Convert with com.sun.javacard.converter.Converter
   - set applet and package AID
3. Verify with com.sun.javacard.offcardverifier.Verifier
   - Java compiler will not catch Java Card restrictions
   - often problems with implicit intermediate data types

Preconfigured ant task: Build

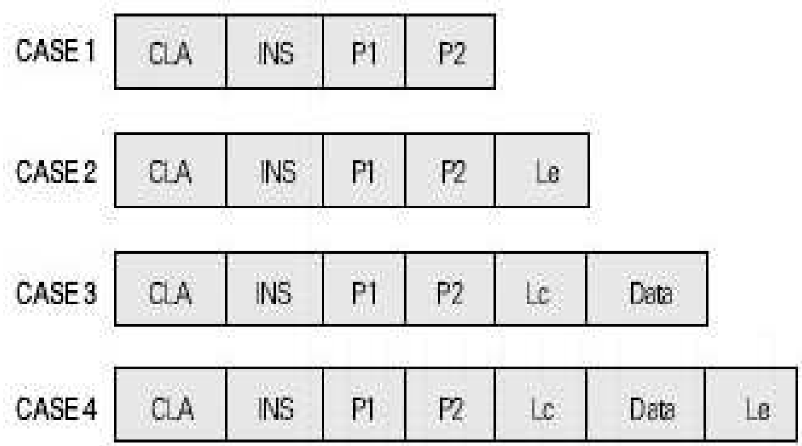
Simple Java Card applet – upload&install

1. Upload and install converted *.cap file
   - GPShell tool with script specific for target card
   - GP SCP channel version (mode_201, mode_211)
   - select CardManager by AID (various AIDs)
   - authenticate and open secure channel (open_sc)
   - delete previous applet version (1. applet, 2. package)
   - load and install (install command, many params)
   - install may pass personalization data (master key…)
2. Initialize applet and check its functionality
   - from GPShell script, no need for secure channel
   - select your applet by AID (select –AID xxx)
   - send test or intialization APDUs (send_apdu -APDU xxx)

Preconfigured ant task: Test

Simple Java Card applet - settings

- Package AID
  - 0x53:0x69:0x6d:0x70:0x6c:0x65:0x50:0x49:0x4e
- Applet AID
  - 0x53:0x69:0x6d:0x70:0x6c:0x65:0x50:0x49:0x4e:0x01
- incData() method
  - CLA = 0xB0
  - INS = 0x10
  - P1 = my number to increase
  - P2 = unused
  - LC = set by terminal
  - Data = send by terminal, LC bytes



Sending and receiving data (in JC applet)

- javacard.framework.APDU
  - incoming and outgoing data in APDU object
  - received inside `process()` method
- Obtaining just apdu header
  - APDU::getBuffer()
  - use to decide what method should be called
- Receive data from terminal
  - APDU::setIncomingAndReceive()
- Send outgoing data
  - APDU::setOutgoingAndSend()

Sending and receiving data – source code

```
private void ReceiveSendData(APDU apdu) {
    byte[]    apdubuf = apdu.getBuffer();     // Get just APDU header (5 bytes)
    short     dataLen = apdu.setIncomingAndReceive(); // Get all incoming data
    // DO SOMETHING WITH INPUT DATA
    // STARTING FROM apdubuf[ISO7816.OFFSET_CDATA]
    // ...
    // FILL SOMETHING TO OUTPUT (apdubuf again), 10 BYTES
    Util.arrayFillNonAtomic(apdubuf, ISO7816.OFFSET_CDATA, 10, (byte) 1);
    // SEND OUTGOING BUFFER
    apdu.setOutgoingAndSend(ISO7816.OFFSET_CDATA, 10);
}
```

select() method

- Method called when applet is set as active
  - for subsequent APDU commands
  - begin of the session
  - use for session data init (clear keys, reset state…)

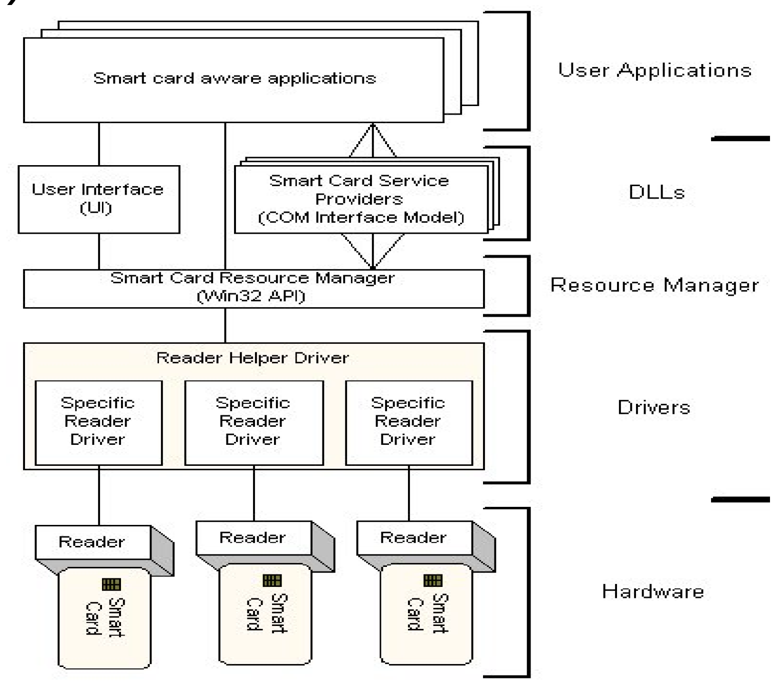
```
public void select() { // CLEAR ALL SESSION DATA
      chv1.reset(); // Reset OwnerPIN verification status
      remainingDataLength = 0; // Set states etc.
      // If card is not blocked, return true.
      // If false is returned, applet is not selectable
     if (!blocked) return true;
     else return false;
  }
```

- deselect()
  - similar, but when applet usage finish
  - may not be called (sudden power drop)

Communication with smart card

How to communicate with our applet?

- Various existing tools for APDU sending
  - e.g., GPShell and send_apdu command
- Possibility to send APDU from our own program
  - PC/SC standard (PC/SC-lite on Linux)
  - SCardxxx Win32 API (winscard.dll)
  - javax.smartcardio.* API for Java 6



Java javax.smartcardio.* API

1. List readers available in system
   a. TerminalFactory::terminals()
   b. identified by index CardTerminal::get(index)
   c. readable string (Gemplus GemPC Card Reader 0)
2. Connect to target card
   a. Check for card (CardTerminal::isCardPresent())
   b. connect to Card (CardTerminal::connect("*"))
   c. get channel (Card::getBasicChannel())
   d. reset card and get ATR (Card::getATR())

Preconfigured project: SimplePIN

Java javax.smartcardio.* API (2)

1. Select applet on card
   a. send APDU with header `00 a4 04 00` *`LC APPLET_AID`*
   b. (see below)
2. Send APDU to invoke method
   a. prepare APDU buffer (byte array)
   b. create CommandAPDU from byte array
   c. send CommandAPDU via CardChannel::transmit()
   d. check for response data (getSW1() == 0x61)
   e. read available response data by `00 C0 00 00` *`SW2`*
3. Process response
   a. status should be ResponseAPDU::getSW() == 0x9000
   b. returned data ResponseAPDU::getData()

Developing simple PKI applet

PKI-relevant Java Card API

- Access controlled by PIN
  - javacard.security.OwnerPIN
- Asymmetric cryptography keys
  - javacard.security.KeyPair, PublicKey, PrivateKey
- Digital signatures
  - javacard.security.Signature
- Asymmetric encryption
  - javacard.security.Cipher

PIN verification functionality

- javacard.framework.OwnerPIN
- Management functions (available for “admin”)
  - Create PIN (new OwnerPIN())
  - Set initial PIN value (OwnerPIN::update())
  - Unblock PIN (OwnerPIN:: resetAndUnblock())
- Common usage functions (available to user)
  - Verify supplied PIN (OwnerPIN::check())
  - Check if was verified (OwnerPIN::isValidated())
  - Get remaining tries (OwnerPIN::getTriesRemaining())
  - Set new value (OwnerPIN::update())

PIN code

```
// CREATE PIN OBJECT (try limit == 5, max. PIN length == 4)
OwnerPIN m_pin = new OwnerPIN((byte) 5, (byte) 4);
// SET CORRECT PIN VALUE
m_pin.update(INIT_PIN, (short) 0, (byte) INIT_PIN.length);
// VERIFY CORRECTNESS OF SUPPLIED PIN
boolean correct = m_pin.check(array_with_pin, (short) 0, (byte)
array_with_pin.length);
// GET REMAING PIN TRIES
byte j = m_pin.getTriesRemaining();
// RESET PIN RETRY COUNTER AND UNBLOCK IF BLOCKED
m_pin.resetAndUnblock();
```

Digital signature

- Management functions
  - Generate new key pair (KeyPair()::genKeyPair())
  - Export public key (KeyPair()::getPublic())
  - (export private key) (KeyPair()::getPrivate())
  - create Signature object (Signature::getInstance())
  - init with public/private key (Signature::init())
- Common usage functions
  - sign message (Signature::update(), Signature::sign())
  - verify signature (Signature::update(),verify())

On-card asymmetric key generation

- javacard.security.KeyPair
- Key pair is generated directly on smart card
  - very good entropy source (TRNG)
  - private key never leaves the card (unless you allow in code)
  - fast sign/verify operation
- But who is sending data to sign/decrypt?
  - protect signature method by PIN::isValidated() check
  - use secure channel to prevent injection of attacker’s message
  - terminal still must be trustworthy

Key generation - source code

```
// CREATE RSA KEYS AND PAIR
m_privateKey = KeyBuilder.buildKey(KeyBuilder.TYPE_RSA_PRIVATE,
     KeyBuilder.LENGTH_RSA_1024,false);
m_publicKey = KeyBuilder.buildKey(KeyBuilder.TYPE_RSA_PUBLIC,
     KeyBuilder.LENGTH_RSA_1024,true);
m_keyPair = new KeyPair(KeyPair.ALG_RSA, (short)
     m_publicKey.getSize());

// STARTS ON-CARD KEY GENERATION PROCESS
m_keyPair.genKeyPair();
// OBTAIN KEY REFERENCES
m_publicKey = m_keyPair.getPublic();
m_privateKey = m_keyPair.getPrivate();
```

Public (private) key export/import

- Obtain algorithm-specific key object from KeyPair
  - e.g., RSAPublicKey pubKey = keyPair.getPublic();
  - get exponent and modulus
    - getExponent() & getModulus() methods
  - send it back to terminal via APDU
- Similar situation with key import
  - setExponent() & setModulus() methods
- Private key export
  - up to you if your code will allow that (usually not)
  - same as public for RSAPublicKey
  - more parameters with RSAPrivateCrtKey (CRT mode)

javacard.security.Signature

- Both symmetric and asymmetric crypto signatures
  - RSA_SHA_PKCS1 (always), ECDSA_SHA (JCOP), DSA (uncommon)
  - DES_MAC8_NOPAD (always), ISO9797 (common), AES (increasingly common)
  - check in advance what your card supports
- Message hashing done on card (asymmetric sign)
  - message received in single or multiple APDUs
  - Signature::update(), Signature::sign()
- If you need just sign of message hash
  - use Cipher object to perform asymmetric crypto operation

Signature – source code

```
// CREATE SIGNATURE OBJECT
Signature m_sign = Signature.getInstance(Signature.ALG_RSA_SHA_PKCS1, false);
// INIT WITH PRIVATE KEY
m_sign.init(m_privateKey, Signature.MODE_SIGN);

// SIGN INCOMING BUFFER
signLen = m_sign.sign(apdubuf, ISO7816.OFFSET_CDATA, (byte) dataLen,
                      m_ramArray, (byte) 0);
```

Asymmetric encryption

- javacardx.crypto.Cipher
- Usage similar to Signature object
  - generate key pair
  - export/import public key
  - initialize Key and set mode (MODE_ENCRYPT/DECRYPT)
  - process incoming data (Cipher::update(), doFinal())
- Supported algorithms
  - RSA_NOPAD (always), RSA_PKCS1 (almost always), EC (sometimes)
- Usable also for symmetric crypto algorithms (later)

Demo - symmetric cryptography applet

Random numbers

- javacard.security.RandomData
- Two versions of random generator
  - ALG_SECURE_RANDOM (truly random)
  - ALG_PSEUDO_RANDOM (deterministic from seed)
- Generate random block
  - RandomData::generateData()
- Very fast and high quality output
  - bottleneck is usually card-to-terminal link

RandomData – source code

```
private RandomData     m_rngRandom = null;
// CREATE RNG OBJECT
m_rngRandom = RandomData.getInstance(RandomData.ALG_SECURE_RANDOM);
// GENERATE RANDOM BLOCK WITH 16 BYTES
m_rngRandom.generateData(array, (short) 0, ARRAY_ONE_BLOCK_16B);
```

Key generation and initialization

- Allocation and initialization of the key object (KeyBuilder.buildKey())
- Receive (or generate random) key value
- Set key value (DESKey.setKey())

```
// …. INICIALIZATION SOMEWHERE (IN CONSTRUCT)
// CREATE DES KEY OBJECT
DESKey m_desKey = (DESKey) KeyBuilder.buildKey(KeyBuilder.TYPE_DES,
   KeyBuilder. LENGTH_DES3_3KEY, false);
// Generate from RNG
m_rngRandom.generateData(array, (short) 0,
   (short) KeyBuilder. LENGTH_DES3_3KEY/8);

// SET KEY VALUE
m_desKey.setKey(array, (short) 0);
```

Symmetric cryptography encryption

- javacard.security.Cipher
- Allocate and initialize cipher object
  - Cipher::getInstance(), Cipher::init()
- Encrypt or decrypt data
  - Cipher.update(), Cipher.doFinal()

Encryption – source code

```
// INIT CIPHER WITH KEY FOR ENCRYPT DIRECTION
m_encryptCipher.init(m_desKey, Cipher.MODE_ENCRYPT);
//….

// ENCRYPT INCOMING BUFFER
void Encrypt(APDU apdu) {
    byte[]   apdubuf = apdu.getBuffer();
    short    dataLen = apdu.setIncomingAndReceive();

    // CHECK EXPECTED LENGTH (MULTIPLY OF 64 bites)
    if ((dataLen % 8) != 0) ISOException.throwIt(SW_CIPHER_DATA_LENGTH_BAD);

    // ENCRYPT INCOMING BUFFER
    m_encryptCipher.doFinal(apdubuf, ISO7816.OFFSET_CDATA, dataLen, m_ramArray,
(short) 0);

    // COPY ENCRYPTED DATA INTO OUTGOING BUFFER
    Util.arrayCopyNonAtomic(m_ramArray, (short) 0, apdubuf, ISO7816.OFFSET_CDATA,
dataLen);

    // SEND OUTGOING BUFFER
    apdu.setOutgoingAndSend(ISO7816.OFFSET_CDATA, dataLen);
}
```

Message authentication code (MAC)

- javacard.security.Signature
- Usage similar to asymmetric signatures
- Create signature object for target MAC algorithm
- Initialize with symmetric cryptography key
- Supported algorithms
  - DES_MAC8 (always), AES_MAC8 (increasingly common)

MAC – source code

```
private   Signature      m_sessionCBCMAC = null;
private   DESKey          m_session3DesKey = null;

// CREATE SIGNATURE OBJECT
m_sessionCBCMAC = Signature.getInstance(Signature.ALG_DES_MAC8_NOPAD, false);
// CREATE KEY USED IN MAC
m_session3DesKey = (DESKey) KeyBuilder.buildKey(KeyBuilder.TYPE_DES,
KeyBuilder.LENGTH_DES3_3KEY, false);

// INITIALIZE SIGNATURE DES KEY
m_session3DesKey.setKey(m_ram, (short) 0);
// SET KEY INTO SIGNATURE OBJECT
m_sessionCBCMAC.init(m_session3DesKey, Signature.MODE_SIGN);

// GENERATE SIGNATURE OF buff ARRAY, STORE INTO m_ram ARRAY
m_sessionCBCMAC.sign(buff, ISO7816.OFFSET_CDATA, length, m_ram, (short) 0);
```

Data hashing

- javacard.security.MessageDigest
- Create hashing object for target algorithm
  - MessageDigest.getInstance()
- Reset internal state of hash object
  - MessageDigest::reset()
- Process all parts of data
  - MessageDigest::update()
- Compute final hash digest
  - MessageDigest.doFinal()
- Supported algorithms
  - MD5, SHA-1 (always), SHA-256 (increasingly common)
  - related to supported Signature algorithms

Data hashing – source code

```
// CREATE SHA-1 OBJECT
MessageDigest m_sha1 = MessageDigest.getInstance(
  MessageDigest.ALG_SHA, false);

// RESET HASH ENGINE
m_sha1.reset();
// PROCESS ALL PARTS OF DATA
while (next_part_to_hash_available) {
 m_sha1.update(array_to_hash, (short) 0, (short) array_to_hash.length);
}
// FINALIZE HASH VALUE (WHEN LAST PART OF DATA IS AVAILABLE)
// AND OBTAIN RESULTING HASH VALUE
m_sha1.doFinal(array_to_hash, (short) 0, (short) array_to_hash.length,
  out_hash_array, (short) 0);
```

What if required algorithm is not supported?

- JavaCard API is limited
  - not all algorithms from standard are supported by particular card
- Own implementation can be written (bytecode)
- Expect much lower performance
  - bytecode interpreted by JCVM
- Expect lower resilience against attacks
  - side channel, fault induction…
- Still doable, see (AES, SHA2-512, OAEP) http://www.fi.muni.cz/~xsvenda/jcalgs

Demo: OpenPGP applet

OpenPGP

- Standard for PGP/GPG compliant applications
- Includes specification for card with private key(s)
  - openpgp-card-1.0.pdf
- Supported (to some extend) in GnuPG
- Pre-personalized OpenPGP cards available
  - http://www.g10code.de/p-card.html
- Open source Java Card applet available
  - JOpenPGPCard
  - http://sourceforge.net/projects/jopenpgpcard/
  - our card can be used

JOpenPGPCard applet

- Main parts
  - two level of PIN protection
  - on-card keys generation, public key export
  - on-card encryption/signature
- Compilation and upload
  - Project settings (preconfigured)
  - AID (given in OpenPGP specification)
  - GPShell script
- Compile and upload applet to card

GPShell script

```
# Install & configure script for Gemalto TOP IM GX4, mother key
mode_201
gemXpressoPro
enable_trace
establish_context
card_connect

select -AID A000000018434D00
open_sc -security 3 -keyind 0 -keyver 0 -key 47454d4585052455353534f53414d504c45

delete -AID ${jc.applet.AID_GPShell}
delete -AID ${jc.package.AID_GPShell}

install -file ${jc.package.shortName}.cap -sdAID A000000018434D00
    -nvCodeLimit 4000 -priv 0

# test selection
select -AID ${jc.applet.AID_GPShell}

card_disconnect
release_context
```

Connect to reader and card

Select Card Manager application

Authenticate and establish secure channel (OpenPlatform)

Delete previous version of our applet (instance first, package second)

Upload and install file *.cap with applet

Try to select newly installed applet

Compilation and upload

- `gpg --card-edit`
- `Command> admin`
- `Command> help`
- `Command> generate`
  - *follow the instructions (default PINs)*
  - signature, decryption and authentication key
  - private keys generated directly on the card
  - public keys exported to GPG keyring
- Change your PIN by `Command> passwd`

GPG --card-edit

europen@europen: ~

File Edit View Search Terminal Help

```
europen@europen:~$ gpg --card-edit

gpg: detected reader `SCM SDI 010 [Vendor Interface] (21120837200398) 00 00'
gpg: detected reader `SCM SDI 010 [Vendor Interface] (21120837200398) 00 01'
Application ID ...: D276000124010101FFFF000000010000
Version ..........: 1.1
Manufacturer .....: test card
Serial number ....: 00000001
Name of cardholder: [not set]
Language prefs ...: [not set]
Sex ..............: unspecified
URL of public key : [not set]
Login data .......: [not set]
Signature PIN ....: forced
Key attributes ...: 1024R 1024R 1024R
Max. PIN lengths .: 32 32 32
PIN retry counter : 3 3 3
Signature counter : 0
Signature key ....: [none]
Encryption key....: [none]
Authentication key: [none]
General key info..: [none]

gpg/card>
```

No keys generated yet

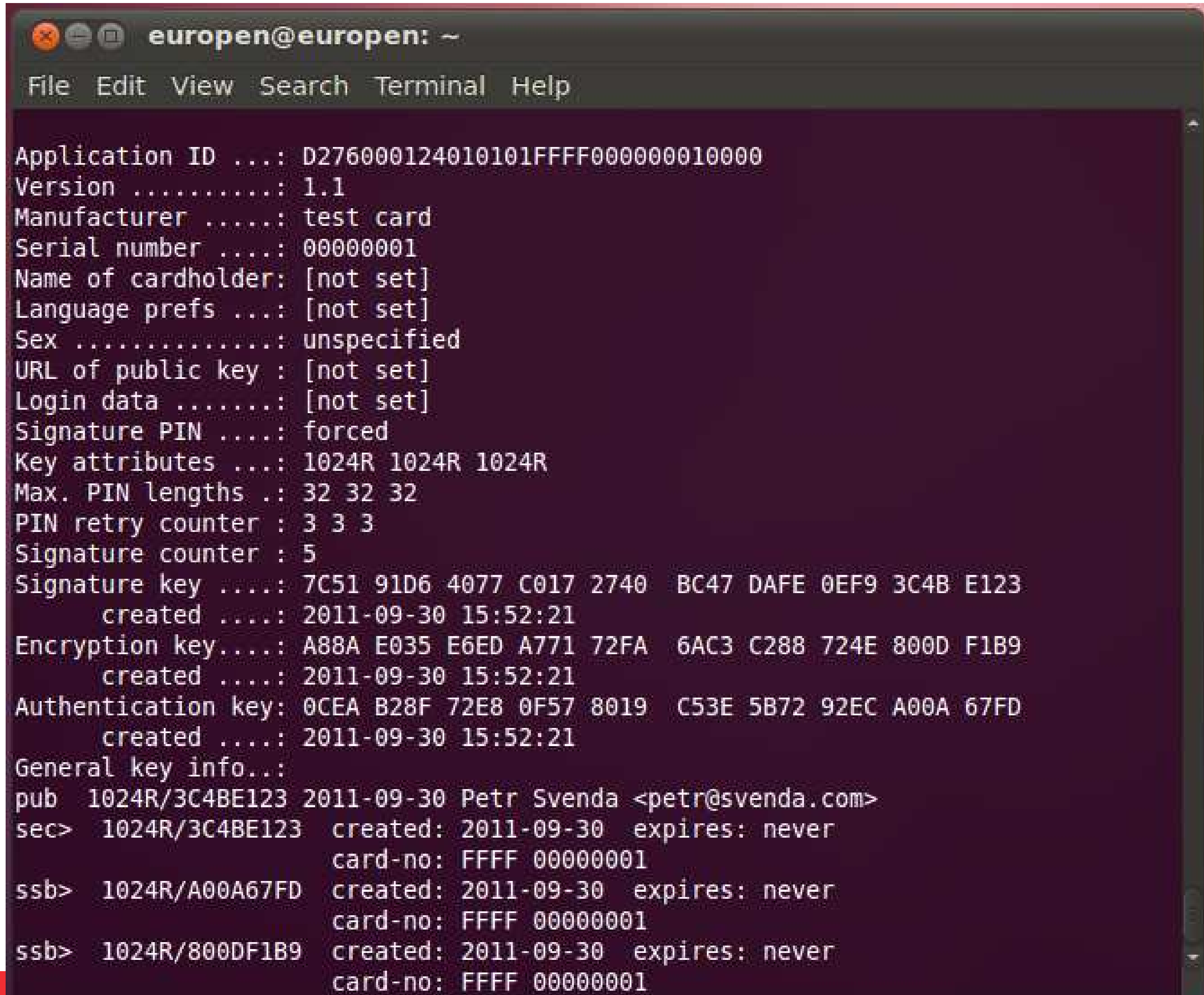
GPG – keys generation finished

```
europen@europen: ~
File  Edit  View  Search  Terminal  Help
gpg: generating new key
gpg: please wait while key is being generated ...
gpg: key generation completed (1 seconds)
gpg: signatures created so far: 1
gpg: signatures created so far: 2
gpg: generating new key
gpg: please wait while key is being generated ...
gpg: key generation completed (2 seconds)
gpg: signatures created so far: 3
gpg: signatures created so far: 4
gpg: key 3C4BE123 marked as ultimately trusted
public and secret key created and signed.

gpg: checking the trustdb
gpg: 3 marginal(s) needed, 1 complete(s) needed, PGP trust model
gpg: depth: 0  valid:   1  signed:   0  trust: 0-, 0q, 0n, 0m, 0f, 1u
pub   1024R/3C4BE123 2011-09-30
      Key fingerprint = 7C51 91D6 4077 C017 2740  BC47 DAFE 0EF9 3C4B E123
uid                  Petr Svenda <petr@svenda.com>
sub   1024R/A00A67FD 2011-09-30
sub   1024R/800DF1B9 2011-09-30


gpg/card>
```

What we have…

- Card with OpenPGP-compliant applet
- GPG generated private&public keypairs
  - sign, enc, auth
  - 1024 bits RSA keys
- Public keys exported from card and imported to local keyring
- Can be used to sign, encrypt message on command line
- Can be further integrated into applications
  - Thunderbird + Enigmail + GPG

(gpg –card-edit) Command> list




```
Application ID ...: D276000124010101FFFF000000010000
Version ..........: 1.1
Manufacturer .....: test card
Serial number ....: 00000001
Name of cardholder: [not set]
Language prefs ...: [not set]
Sex ..............: unspecified
URL of public key : [not set]
Login data .......: [not set]
Signature PIN ....: forced
Key attributes ...: 1024R 1024R 1024R
Max. PIN lengths .: 32 32 32
PIN retry counter : 3 3 3
Signature counter : 5
Signature key ....: 7C51 91D6 4077 C017 2740  BC47 DAFE 0EF9 3C4B E123
      created ....: 2011-09-30 15:52:21
Encryption key....: A88A E035 E6ED A771 72FA  6AC3 C288 724E 800D F1B9
      created ....: 2011-09-30 15:52:21
Authentication key: 0CEA B28F 72E8 0F57 8019  C53E 5B72 92EC A00A 67FD
      created ....: 2011-09-30 15:52:21
General key info..:
pub  1024R/3C4BE123 2011-09-30 Petr Svenda <petr@svenda.com>
sec>  1024R/3C4BE123  created: 2011-09-30  expires: never
                      card-no: FFFF 00000001
ssb>  1024R/A00A67FD  created: 2011-09-30  expires: never
                      card-no: FFFF 00000001
ssb>  1024R/800DF1B9  created: 2011-09-30  expires: never
                      card-no: FFFF 00000001
```

Using GPG with smart card

- **`gpg --clearsign --output myfile.sig --sign myfile`**
  - our public key is already imported to keyring
  - PIN is required to sign (notice signature count so far)
  - --clearsign causes output in BASE64
- **`gpg --verify myfile.sig`**
  - smart card not required, public key in keyring
- **`gpg --output gpshell.log.gpg --recipient petr@svenda.com --encrypt gpshell.log`**
  - smart card not required, public key in keyring
- **`gpg --decrypt gpshell.log.gpg`**

Demo: CardEdge applet

PKCS#11, PKCS#15, ISO/IEC 7816-15

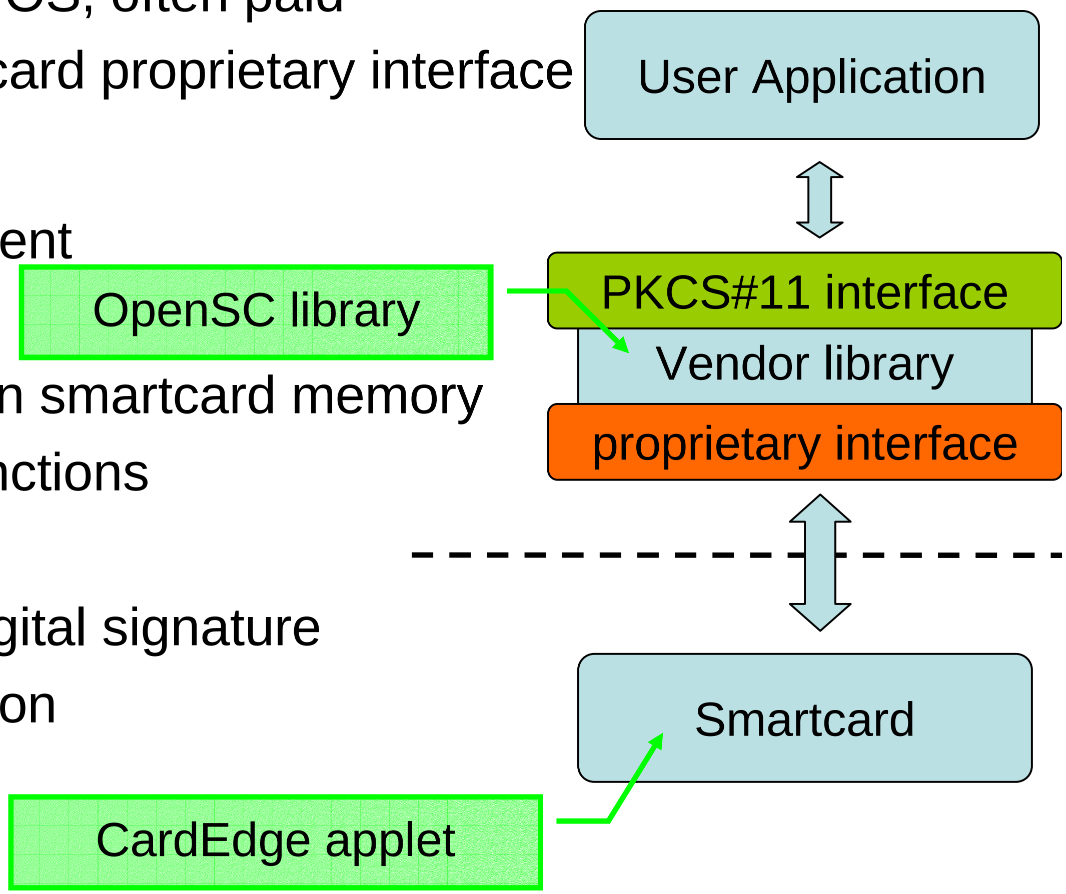
- Standards for API of cryptographic tokens
- PKCS#11
  - http://www.rsa.com/rsalabs/node.asp?id=2133
  - software library on PC, rather low level functions
  - widely used, TrueCrypt, Firefox, Thunderbird…
- PKCS#15
  - http://www.rsa.com/rsalabs/node.asp?id=2141
  - both hardware and software-only tokens
  - identity cards…
  - superseded by ISO/IEC 7816-15 standard

CardEdge applet

- Main parts
  - multiple different PINs for different objects
  - symmetric cryptography, key management
  - on-card keys generation, public key export
  - on-card encryption/signature…
- Compilation and upload
  - project settings (preconfigured), AID (a00000000101)
  - GPShell script (upload, set default PIN)
- Personalize (pkcs15-init)
  - create PKCS#15 structure (label, PIN, PUK)
  - bash script (preconfigured)

PKCS#11

- Standardized interface of security-related functions
  - vendor-specific library in OS, often paid
  - communication library->card proprietary interface
- Functionality cover
  - slot and token management
  - session management
  - management of objects in smartcard memory
  - encryption/decryption functions
  - message digest
  - creation/verification of digital signature
  - random number generation
  - PIN management



PKCS#15 (OpenSC)

- **`pkcs15-init`**

```
#!/bin/bash

sleep 5
pkcs15-init --create-pkcs15 --pin 12345678 --no-so-pin
sleep 5
pkcs15-init --store-pin --auth-id 01 --pin 12345678
            --puk 12345678 --label "EurOpen Tutorial"
```

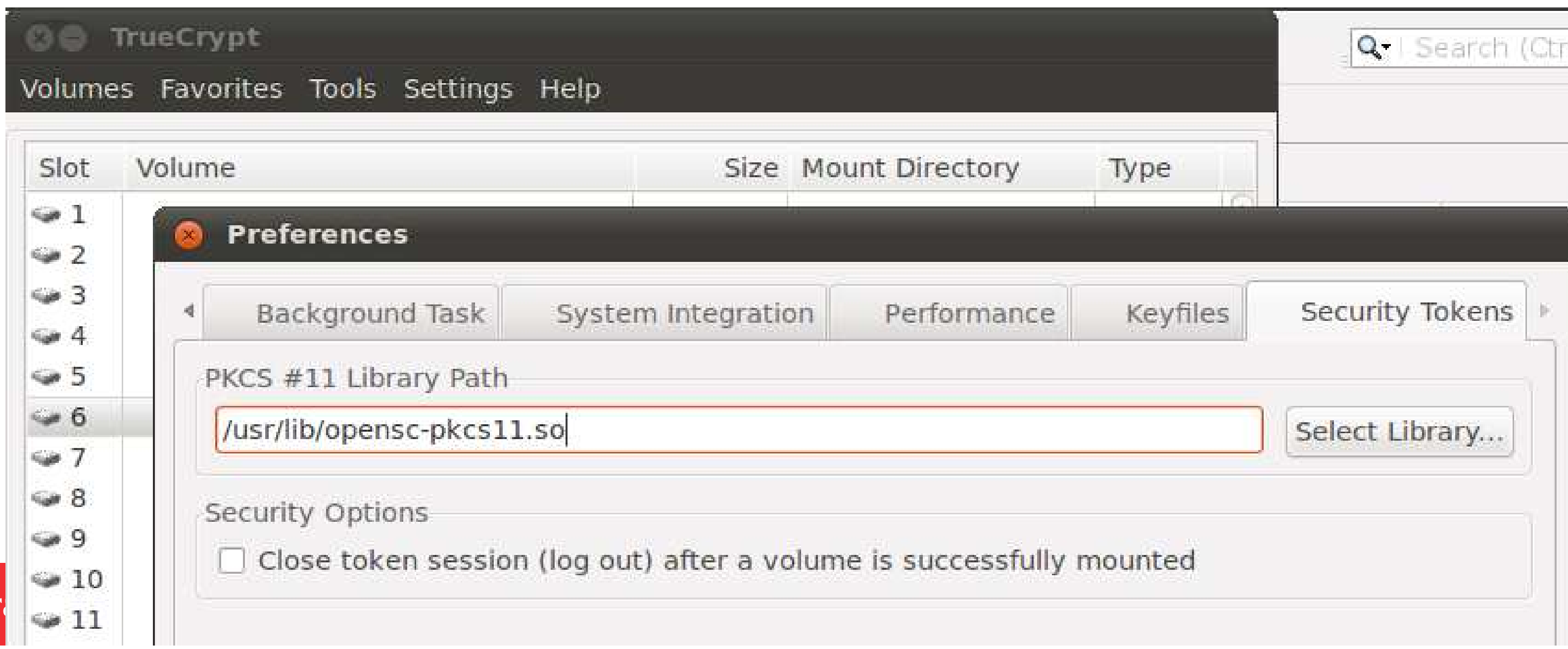
- **`pkcs15-tool --dump`**
- **`pkcs15-tool --list-keys`**

What we have…

- Card with CardEdge (Muscle) applet
  - interface not standardized (as created by Muscle)
  - all functionality provided by card
- OpenSC (http://www.opensc-project.org/opensc)
- OpenSC project providing multiple tools
  - both Windows and Linux
  - able to communicate with CardEdge applet
- OpenSC PKCS#11 (multiple apps, e.g., TrueCrypt)
- OpenSC PKCS#15 (ISO/IEC 7816-15, id cards)

TrueCrypt and PKCS#11 token

- CardEdge applet + OpenSC PKCS#11 library
- TrueCrypt→Settings→Security Tokens…
  - PKCS #11 Library Path
  - /usr/lib/opensc-pkcs11.so [Linux]
  - opensc.dll [Windows]



TrueCrypt and PKCS#11 token (2)

- Create new disk
  - Tools →Volume Creation Wizard
  - Follow instructions until screen with ‘Volume Password’
- Generate Random Keyfile…
  - save random file to disk
- Button KeyFiles…→ Add Token Files…
- ‘Import Keyfile to Token’
  - import file previously saved to disk
  - (backup file and delete from disk)
- Select previously imported token file and confirm
- Continue and finalize TrueCrypt disk generation

TrueCrypt and PKCS#11 token (3)

- Mount→KeyFiles…→Add Token Files…
- Select previously imported token file and confirm
- Automate: ~/truecrypt_mount_volume.sh

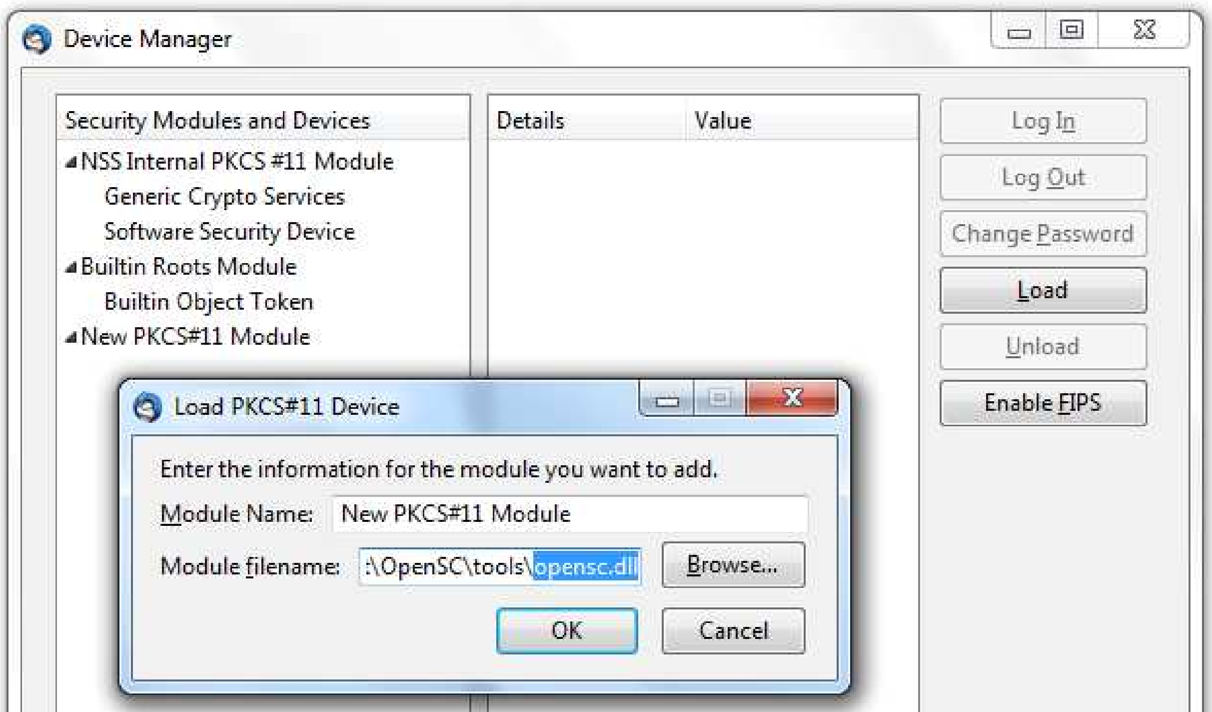
token driver (PKCS#11)

KeyFile on token

```
truecrypt --mount --keyfiles=token://slot/1/file/keyfile
    --token-lib=/usr/lib/opensc-pkcs11.so
    /home/europen/volume1 /media/truecrypt6
```

Thunderbird & S/MIME with PKCS#11

- Tools→Options→Advanced→Security devices
- Load → select pkcs#11 library
  - opensc.dll [Windows]
  - opensc-pkcs11.so [Linux]



Obtain X.509 certificate

- Use your favorite certification authority
  - will provide you with *.p12 file (private key protected by password)
- OR use OpenSSL to generate self-signed cert. (ugly)
  - `openssl genrsa -out my.key 2048`
    - (unable to write 'random state' may appear - not important)
    - my.key file will be created in current directory
  - `openssl req -new -x509 -days 365 -key my.key -out my.crt -sha512`
    - fill in certificate parameters (see next slide for example)
  - `openssl pkcs12 -export -out my.p12 -in my.crt -inkey my.key`
    - export your private and public key into single my.p12 file
    - import later on target machine into certificate store

OpenSSL X.509 certificate info

- Use this file as input for `openssl req -new` …

```
CZ
Czech Republic

Masaryk University
LaBAK
Petr Svenda
svenda@fi.muni.cz



REM !!! newlines are important
```

Thunderbird & S/MIME with PKCS#11

- Import keys from *.p12 to token
  - Account settings→Security→View Certificates→Import
- Setting signature and decryption keys
  - Account settings→Security, Digital signing, Encryption

Best practices

Execution speed hints (1)

- Difference between RAM and EEPROM memory
  - *new* allocates in EEPROM (persistent, but slow)
    - do not use EEPROM for temporary data
    - do not use for sensitive data (keys)
  - JCSystem::getTransientByteArray() for RAM buffer
  - local variables automatically in RAM
- Use API algorithms and utility methods
  - much faster, cryptographic co-processor
- Allocate all resources in constructor
  - executed during installation (only once)
  - either you get everything you want or not install at all

Execution speed hints (2)

- Garbage collection may not be available
  - do not use *new* except in constructor
- Keep Cipher or Signature objects initialized
  - if possible (e.g., fixed master key)
  - initialization with key takes non-trivial time
- Use copy-free style of methods
  - foo(byte[] buffer, short start_offset, short length)
- Do not use recursion or frequent function calls
  - slow, function context overhead
- Do not use OO design extensively (slow)

Security hints (1)

- Use API algorithms/modes rather than your own
  - API algorithms fast and protected in cryptographic hardware
  - general-purpose processor leaking more information
- Store session data in RAM
  - faster and more secure against power analysis
  - EEPROM has limited number of rewrites (10^5 - 10^6 writes)
- Never store keys and PINs in primitive arrays
  - use specialized objects like OwnerPIN and Key
  - better protected against power, fault and memory read-out attacks

Security hints (2)

- Erase unused keys and sensitive arrays
  - use specialized method if exists (Key::clearKey())
  - or overwrite with random data (Random::generate())
- Use transactions to ensure atomic operations
  - power supply can be interrupted inside code execution
  - be aware of attacks by interrupted transactions - rollback attack
- Do not use conditional jumps with sensitive data
  - branching after condition is recognizable with power analysis

Security hints (3)

- Allocate all necessary resources in constructor
  - applet installation usually in trusted environment
  - prevent attacks based on limiting available resources
- Use automata-based programming model
  - well defined states (e.g., user PIN verified)
  - well defined transitions and allowed method calls
- Some additional hints
  - Gemalto_JavaCard_DevelGuide.pdf
  - http://developer.gemalto.com/fileadmin/contrib/downloads/pdf/Java%20Card%20%26%20STK%20Applet%20Development%20Guidelines.pdf

Some practical attacks

Common and realizable attacks

- API-level attacks
  - incorrectly designed and implemented application
- Communication-level attacks
  - observation and manipulation of communication channel
- Side-channel attacks
  - realistic timing and power analysis attacks
- Semi-invasive attacks
  - realistic fault induction attacks

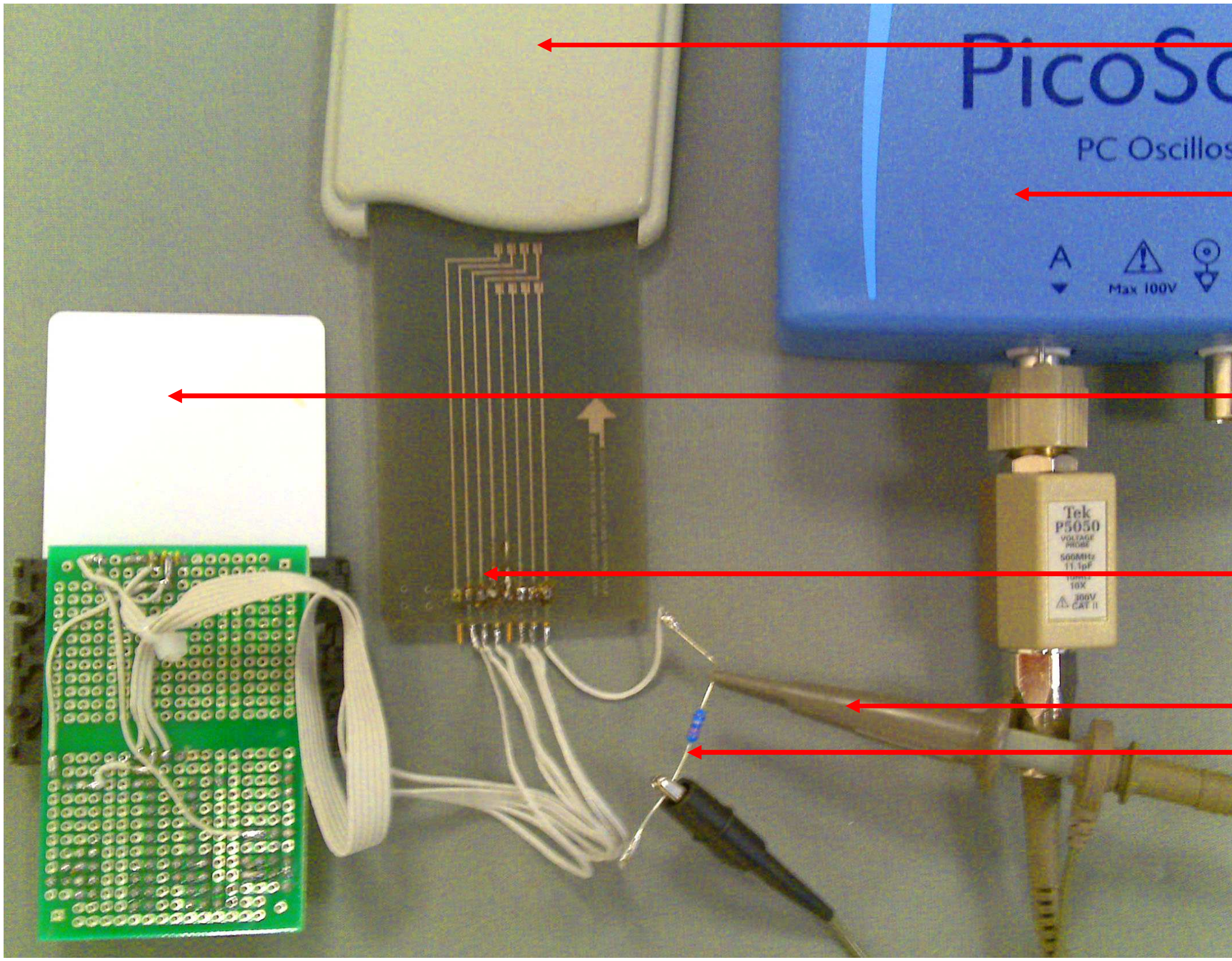
API-level attacks

- Unintentional sensitive function call
  - missing or incoherent authorization
- Unintentional data leakage
  - do not use APDU buffer for internal storage
  - clear memory or select and deselect
- Use automata-based programming
  - well defined states
  - check state before proceeding in function

Communication-level attacks

- Capture data
  - fake library, usb logger, hardware logger, MitM reader
  - use secure channel with encryption
- Modify packets
  - easy to on several levels (same as capture)
  - use secure channel with integrity
- Replay packets
  - use secure channel with MAC chaining
- All threats addressed with GlobalPlatform SCP’0x
  - secure channel protocol

Basic setup for power analysis



Smart card reader

Oscilloscope

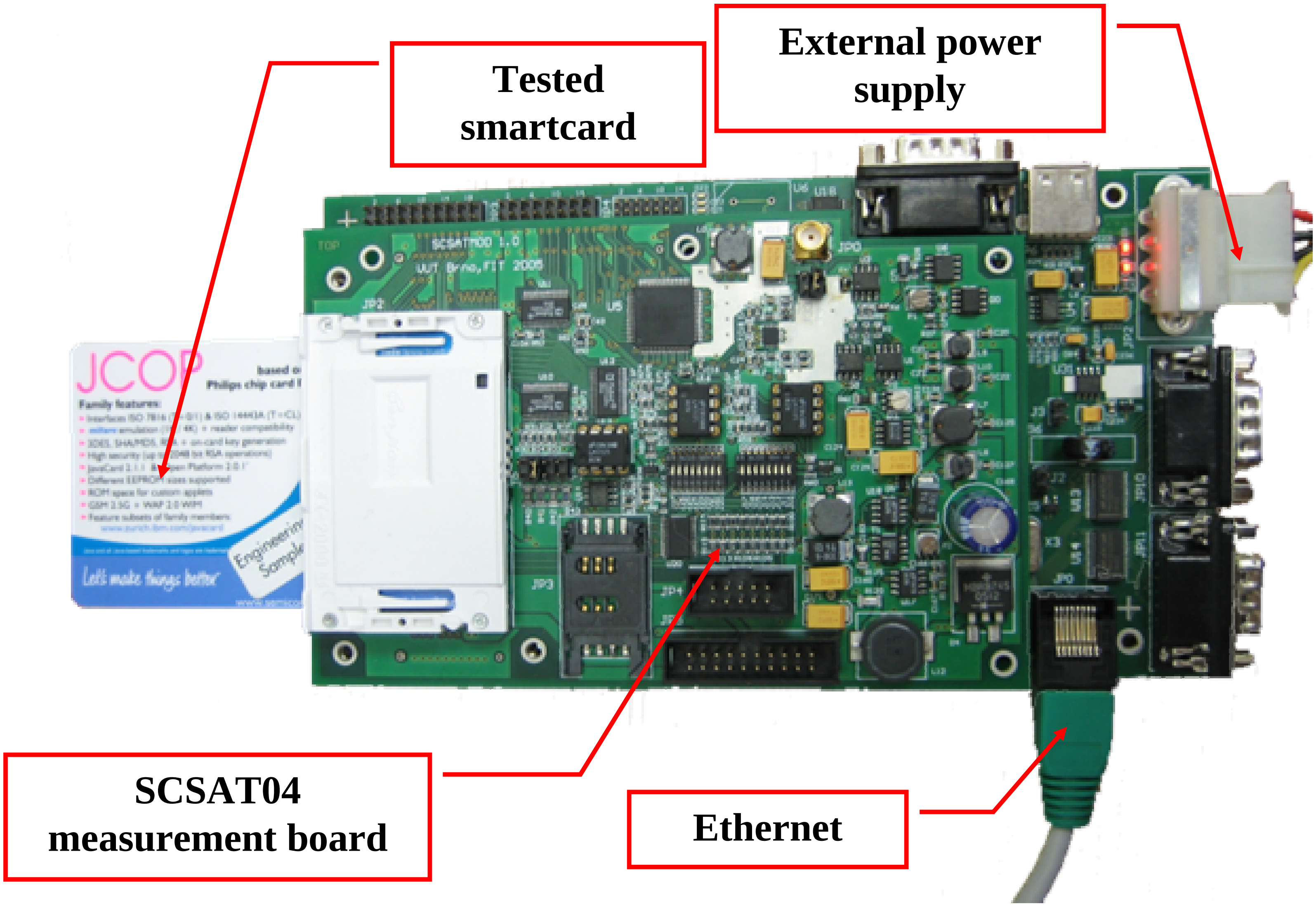
Smart card

Inverse card connector

Probe

Resistor 20-80 ohm

More advanced setup for power analysis



Reverse engineering of Java Card bytecode

- Goal: obtain code back from smart card
  - JavaCard defines around 140 bytecode instructions
  - JVM fetch instruction and execute it

(source code)

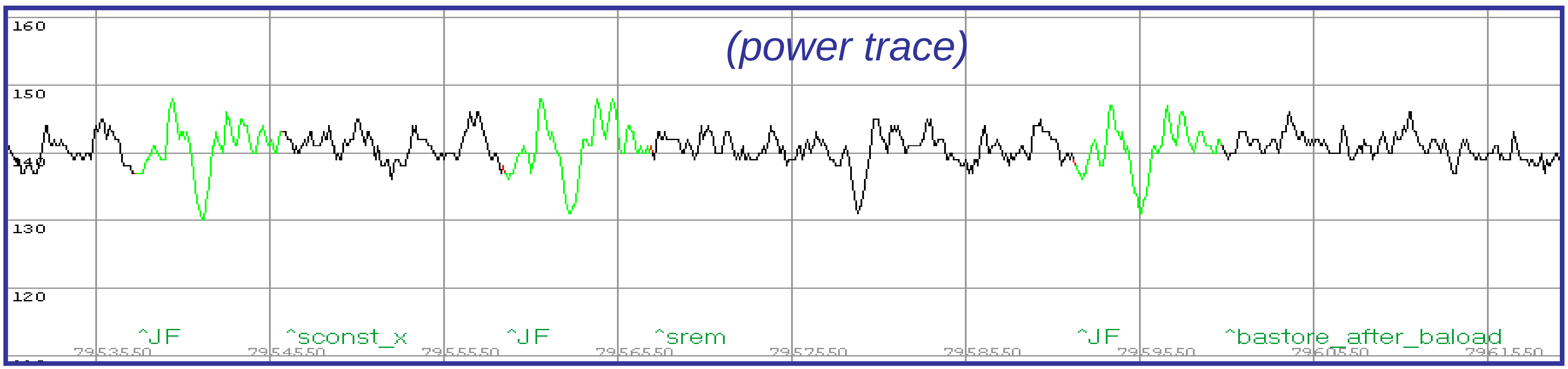
```
m_ram1[0] = (byte) (m_ram1[0] % 1);
```

compiler →

(bytecode)

```
getfield_a_this 0;
sconst_0;
baload;
sconst_1;
srem;
bastore;
```

oscilloscope ↓



Conditional jumps

- may reveal sensitive info
- keys, internal branches, …

(source code)

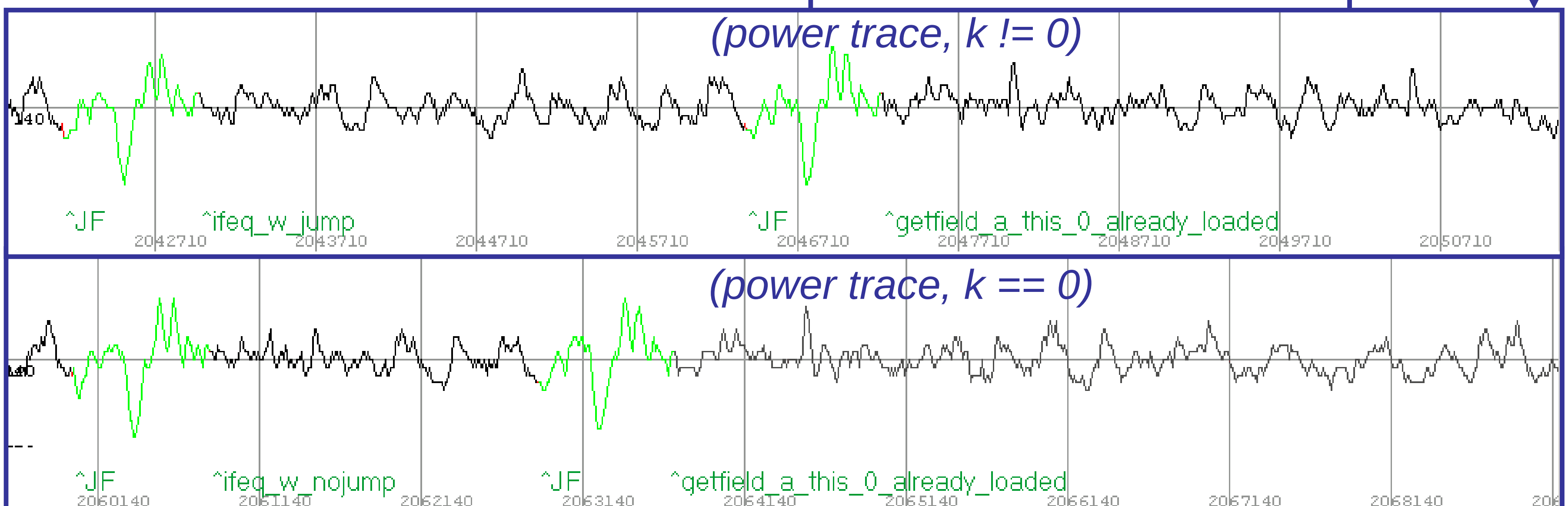
```
if (key == 0) m_ram1[0] = 1;
else m_ram1[0] = 0;
```

compiler

(bytecode)

```
     sload_1;
     ifeq_w L2;
L1:  getfield_a_this 0;
     sconst_0;
     sconst_0;
     bastore;
     goto L3;
L2:  getfield_a_this 0;
     sconst_0;
     sconst_1;
     bastore;
     goto L3;
L3:  …
```

oscilloscope



Semi-invasive attacks

- Physical manipulation, but card still working
  - liquid nitrogen, power glitches, light flashes…
- Fault induction
  - modify memory (RAM, EEPROM), e.g., PIN counter
  - modify instruction, e.g., conditional jump
- Possible protections
  - shadow variable
  - automaton-based execution

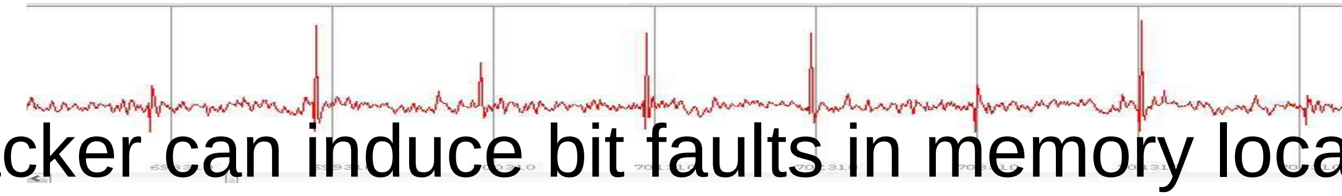
Automated code transformation
CesTa project

- http://cesta.sourceforge.net

Main design goals

1. Enhanced security on real applets
   - fix what is wrong, add preventive defenses
2. Source code level & auditability
   - Trust, but Verify
3. Complexity is hidden
   - clarity of original code
4. Flexibility & Extensibility
   - protect against new threats
   - protect only what HW does not

Another attack – fault induction



- Attacker can induce bit faults in memory locations
  - power glitch, flash light, radiation...
  - harder to induce targeted then random fault
- Protection with shadow variable
  - every variable has *shadow* counterpart
  - shadow variable contains *inverse* value
  - consistency is checked every read/write to memory

01011010

10100101

a	**01011010**	*if (a != ~a_inv) Excepti*	**01010000**	*if (a != ~a_inv) Exception();*
		a = 0x55;		*a = 0x13;*
a_inv	**10100101**	*a_inv = ~0x55;*	**1010101**	

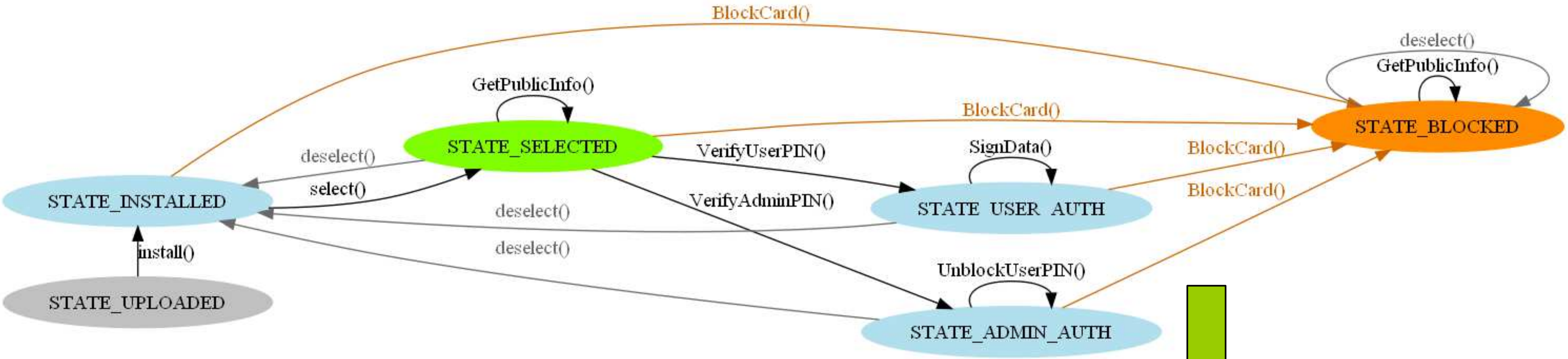
- Robust protection, but cumbersome for developer

Applet state transition enforcement

- Applet security states controlled usually ad-hoc
  - *if (adminPIN.isValidated() && bSecureChannelExists)* …
  - unwanted (unprotected) paths may exist
- Possible solution
  - model state transitions in inspectable format (DOT (GraphViz)
  - automatica
  - check appr

```
digraph StateModel {
rankdir=LR;
size="6,6";
node [shape =ellipse color=lightblue2, style=filled];

{ rank=same; "STATE_UPLOADED";"STATE_INSTALLED";}
"STATE_INSTALLED" [color=lightblue2, style=filled];
"STATE_UPLOADED" [color=gray, style=filled];
"STATE_UPLOADED" -> "STATE_INSTALLED" [label="install()"];
```

Applet state transition - example




```
private void SetStateTransition(short newState) throws Exception {
  // CHECK IF TRANSITION IS ALLOWED
  switch (m_currentState) {
    case STATE_UPLOADED: {
        if (newState == STATE_INSTALLED) {m_currentState = STATE_INSTALLED; break;}
        throw new Exception();
    }
    case STATE_INSTALLED: {
        if (newState == STATE_SELECTED) {m_currentState = STATE_SELECTED; break;}
        if (newState == STATE_BLOCKED) {m_currentState = STATE_BLOCKED; break;}
        throw new Exception();
    }

    case STATE_SELECTED: {
        if (newState == STATE_SELECTED) {m_currentState = STATE_SELECTED; break;}
        if (newState == STATE_USER_AUTH) {m_currentState = STATE_USER_AUTH; break;}
        if (newState == STATE_ADMIN_AUTH) {m_currentState = STATE_ADMIN_AUTH; break;}
        if (newState == STATE_BLOCKED) {m_currentState = STATE_BLOCKED; break;}
        if (newState == STATE_INSTALLED) {m_currentState = STATE_INSTALLED; break;}
```

Check transactions

```
a[0] = 0
beginTransaction()
  a[0] = 1;
  arrayFillNonAtomic(a,0,1,2);
  // a[0] = 2;
abortTransaction()
```

```
a[0] = 0;
beginTransaction();
  arrayFillNonAtomic(a,0,1,2);
  // a[0] = 2;
  a[0] = 1;
abortTransaction();
```

- Transactions can breach applet security
  - e.g., decreased PIN counter value is rolled back
- CesTa can detect possible problems in code
  - warning is generated

```
/***** WARNING *****
         Transaction may contain dangerous operations,
         some variables are used in both assignments and
         non atomic operations: a, b
 ***** WARNING *****/ JCSystem.beginTransaction()/* detected start of transaction */;
a[0] = 1;
b[0] = 2;
Util.arrayFillNonAtomic(a, (short) 0, (short) 1, (byte) 2); // a[0] = 2;
javacard.framework.Util.arrayFillNonAtomic(b, (short) 0, (short) 1, (byte) 2);
JCSystem.abortTransaction()/* detected end of transaction */;
```

CesTa project – current state

- Several non-trivial transformations implemented
  - low level *IfSwitchReplacement* (replacement rule)
  - generic *ShadowVariables* (replacement rule)
  - generic *ValidateStateTransitions* (replacement rule)
  - generic *CheckTransactions* (analysis rule)
- Easy to use and relatively error prone
  - automated unit testing
- Tested on real (bigger) applets
  - JOpenPGPCard, CardCrypt/TrueCrypt, crypto software impl…
- Transformations can be provided by independent labs
  - modular design, open source http://CesTa.sourceforge.net

CesTa project – example

- Project SecureCardEdge is CardEdge applet
  - (NetBeans project in Ubuntu image)
- SecureCardEdge has modified build.xml
  - CesTa transformations are automatically applied
- Integration to existing applets is easy
- Try it ☺

Summary

- Smart cards are programmable (Java Card)
  - reasonable rich cryptographic API
  - coprocessor for fast cryptographic operations
  - multiple applications securely on single card
- PKI applet can be developed with free tools
  - PIN protection, on-card key generation, signature…
  - basic algorithms + programmable extensions
- Standard Java 6 API for smart cards comm.
- Be aware of practical attacks