BUSLab

Brno University Security Laboratory

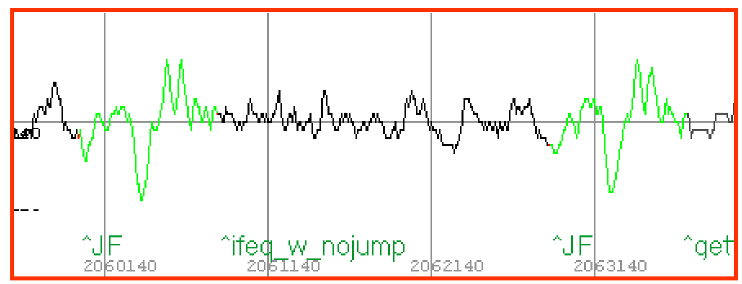
# Automatic source code transformations for strengthening practical security of smart card applications

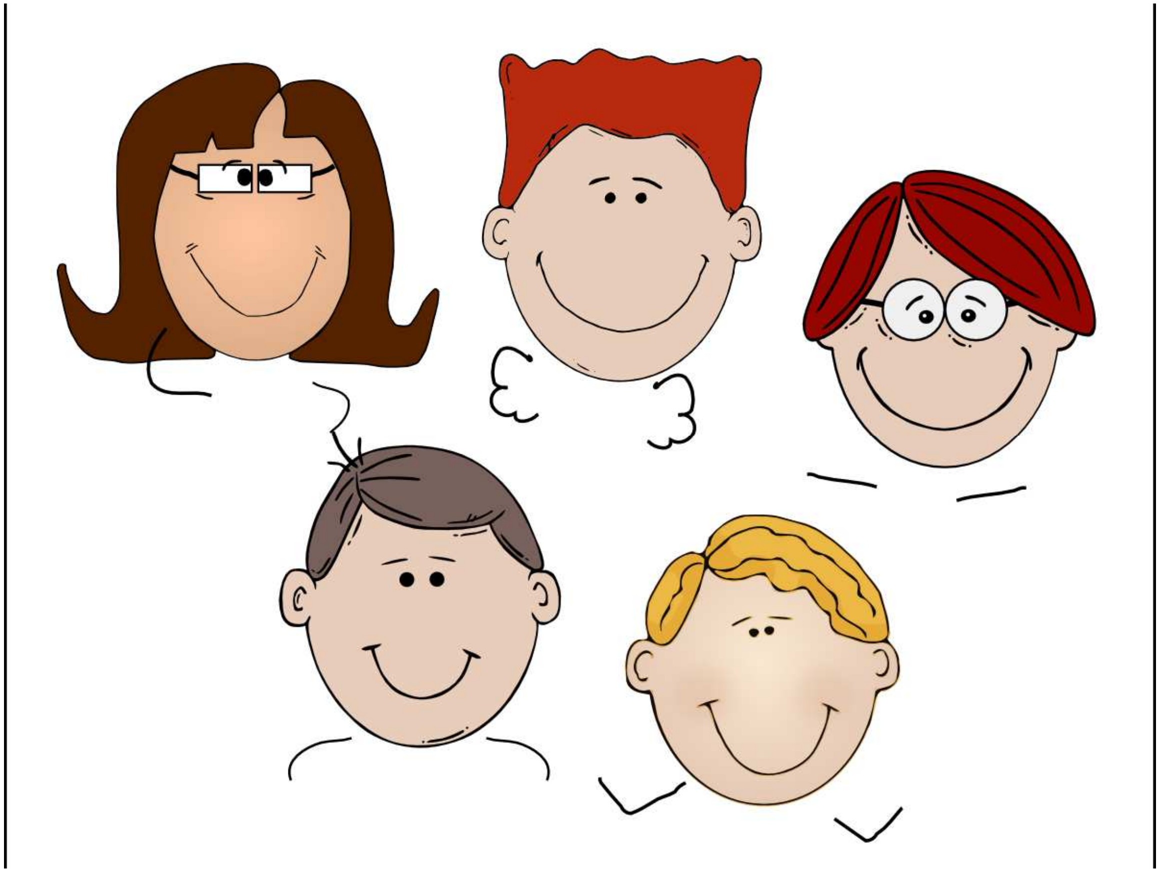
Vašek Lorenc, Tobiáš Smolka and Petr Švenda
Masaryk University, Czech Republic,
*valor@ics.muni.cz, {xsmolka,svenda}@fi.muni.cz*

# Agenda



- Smartcards
- Problems of Java Card platform
- Automated tool for mitigation - CesTa
  - Code Enhancing Security Transformations and Analysis
- Some practical examples
- Wider usage and future work

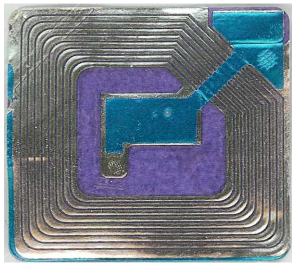




**Happy *Virgin* Java Card friends**

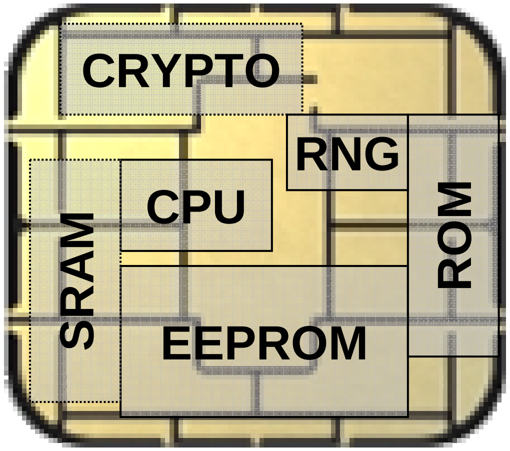
# Some cryptographic smart card facts

# 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)
- 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



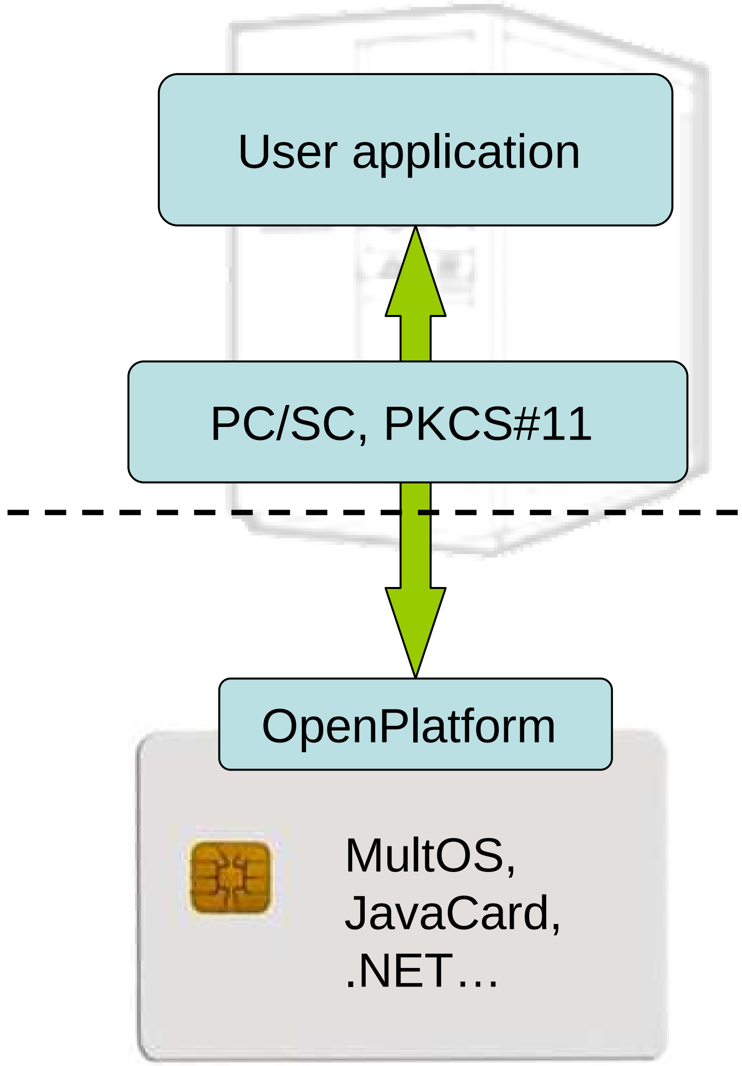

- SC is quite powerful device
  - 8-32 bit procesors @ 5-20MHz
  - persistent memory 32-100kB (EEPROM)
  - volatile fast RAM, usually <<10kB
  - truly random generator
  - cryptographic coprocessor (3DES, RSA-2048,...)
- Programmable (C, *JavaCard*, .NET)
    - (Java) Virtual Machine
    - multiple CPU ticks per bytecode instruction
  - interfaces
    - I/O data line, voltage and GND line (no internal power source)
    - clock line, reset lines
- 5.045 billion units shipped in 2008 (EUROSMART)
  - 4 185 million smartcards, 800 million memory cards
  - 3 580Mu in Telcom, 680Mu payment and loyalty...

# Supported algorithms

- Symmetric cryptography
  - DES, 3DES, AES, RCx (~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!
    - (but who is sending data to sign/decrypt?)
- 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)
- See *http://www.fi.muni.cz/~xsvenda/jcsupport.html* for more

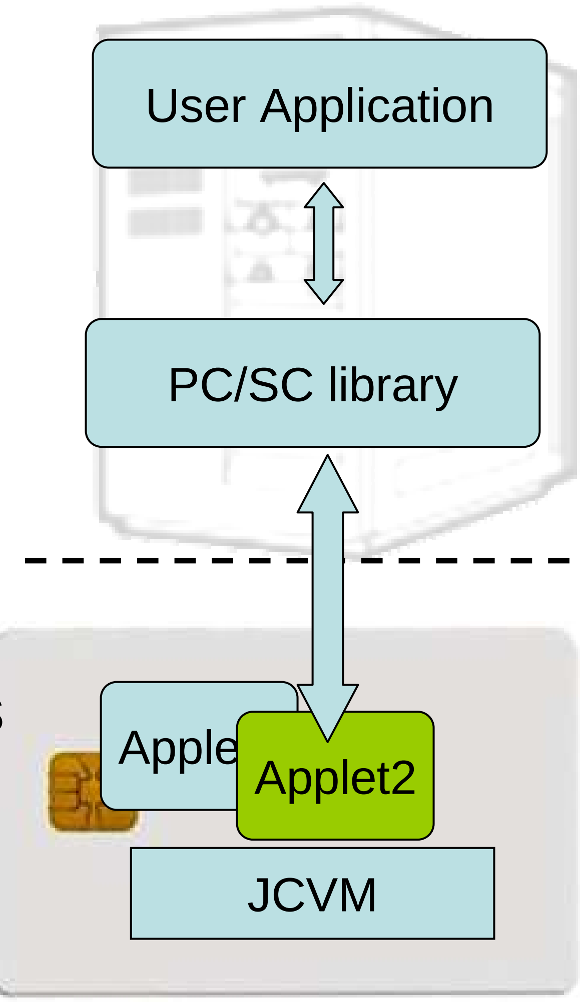
# Common environments and interfaces

- Java Card
  - open programming platform from Sun
  - applets portable between cards
- Microsoft .NET for smartcards
  - relatively new technology
  - similar to Java Card
  - applications portable between cards
- PC/SC, PKCS#11
  - standardized interface on host side
  - card can be proprietary
- OpenPlatform (GlobalPlatform)
  - remote card management interface
  - secure installation of applications



# Java Card 2.x applets

- Writing in restricted Java syntax
- 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 – My first applet

- Desktop Java vs. Java Card
  - PHP vs. C ☺
- Limited type system
  - No ints (*short int* and *byte* only), no floats, no Strings
- No modern programming features
  - No threads, no generics, no iterators…

# Java Card – more to be discovered

- Recursion is slooow...
- Memory allocation issues
  - EEPROM vs. RAM allocations, *new* operator
  - No garbage collector!
- Persistent objects
- Transactions, atomic operations
- Java Card applet firewall

```
function f(…) {
    byte a[] = new byte[10];
    byte b[] = JCSystem.makeTransientByteArray(...);
    byte c;
}
```

# Execution speedup – best practices

- **Avoid EEPROM writes**
  - RAM writes are 1.000 times faster
- Use special functions to manipulate arrays
  - arrayFillNonAtomic(), arrayCopy(), arrayCopyNonAtomic()
- Use native code to perform an operation!
- Use transient arrays to store session and temporary data
- Avoid deep class trees
  - the deeper the tree, the slower the search for virtual methods
- array.length in a local variable when used in a loop
- Exceptions not for flow control, only for error handling

# Java Card – PIN verification

- Image/code for PIN verification
  - Vulnerable to transaction rollback

```
public class OwnerPIN implements PIN {
    byte triesLeft; // persistent counter

    boolean check(...) {
        ...
        triesLeft--;
        ...
    }
}
```

# JavaCard – PIN verification done better

- Non-atomic operations

```
public class OwnerPIN implements PIN {
  byte[] triesLeft = new byte[1]; // persistent counter
  byte[] temps =
     JCSystem.makeTransientByteArray(1,
                JCSystem.CLEAR_ON_RESET);

  boolean check(...) {
    ...
    temps[0] = triesLeft[0] - 1;
    // update the try counter non-atomically:
    Util.arrayCopyNonAtomic(temps, 0, triesLeft, 0, 1);
   ...
  }
}
```

# JavaCard – Atomic vs. Non-Atomic

- Persistent memory updates
  - Two ways of updating
  - FillArrayNonAtomic, CopyArrayNonAtomic
- Code refactoring
  - Original short/byte values have to be converted to arrays[1]

# Java Card – Atomic vs. Non-Atomic

- Non-deterministic variable rollback

```
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();
```

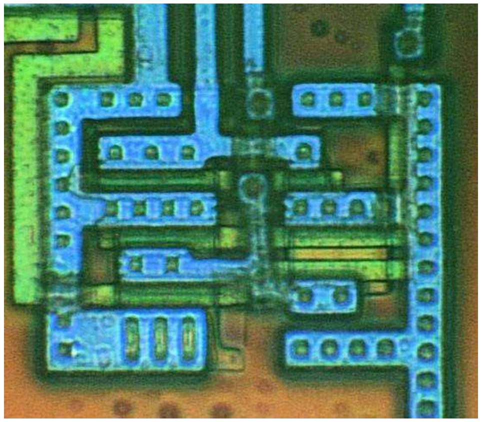
- Result dependency on the commands order
  - **a[0] == 0 vs. a[0] == 2**

# Java Card applet firewall issues

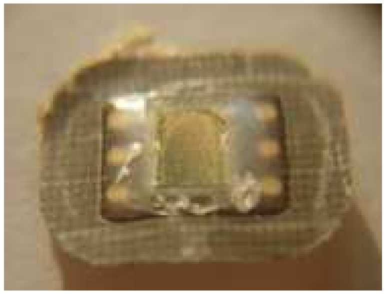
- Main defense for separation of multiple applets
- Platform implementations differ
  - Usually due to the unclear and complex specification
- If problem exists then is out of developer's control
- Firewall Tester project (W. Mostowski)
  - Open and free, the goal is to test the platform

```
short[] array1, array2; // persistent variables
short[] localArray = null; // local array
JCSystem.beginTransaction();
        array1 = new short[1];
        array2 = localArray = array1; // dangling reference!
JCSystem.abortTransaction();
```

# There is more to attack…



- Invasive
  - physical de-packaging, chip often destroyed
  - reading microprobes, direct memory access
  - usually high cost attack



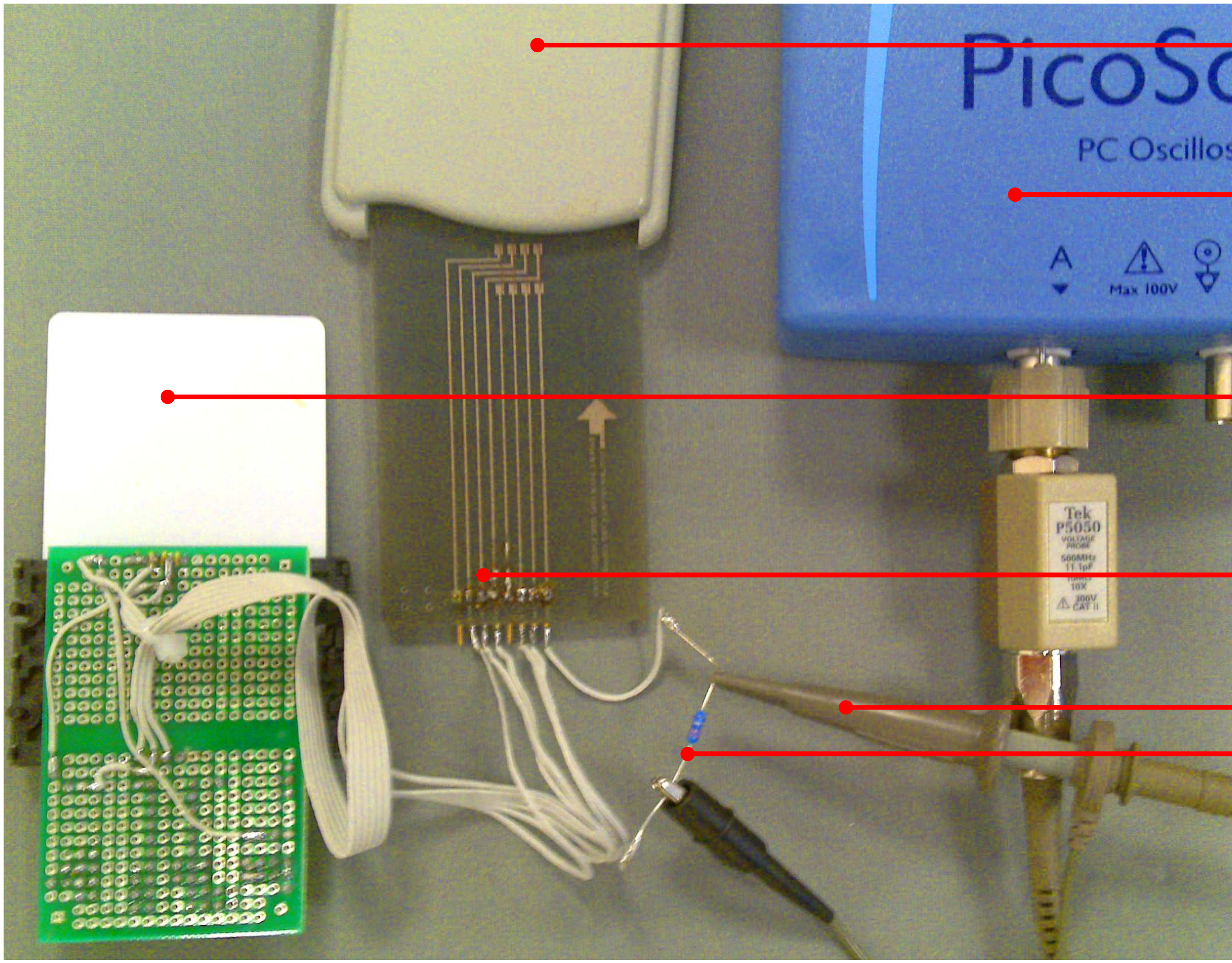
- Semi-invasive
  - often de-packaging, but chip still usable/working
  - optical fault induction
  - supply voltage and clock peaks, …
  - often low cost

- Non-invasive
  - passive observation, chip not affected
  - timing and power analysis, logical API attacks, …

# Smart cards power analysis

- Significant vulnerability exposed
  - external power needed
- Si
- Di
- In

# Basic setup for power analysis




Smart card reader

Oscilloscope

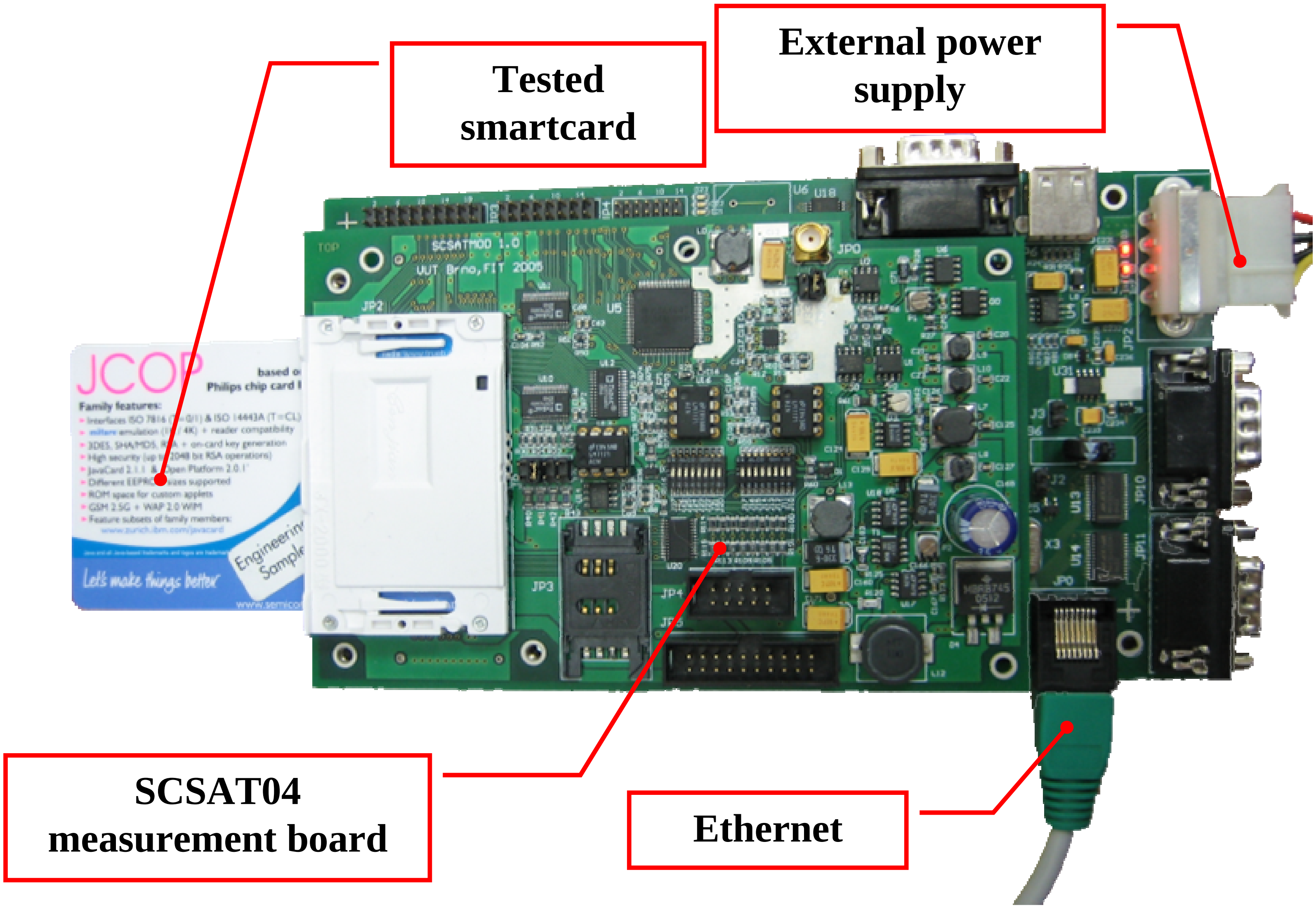
Smart card

Inverse card connector

Probe

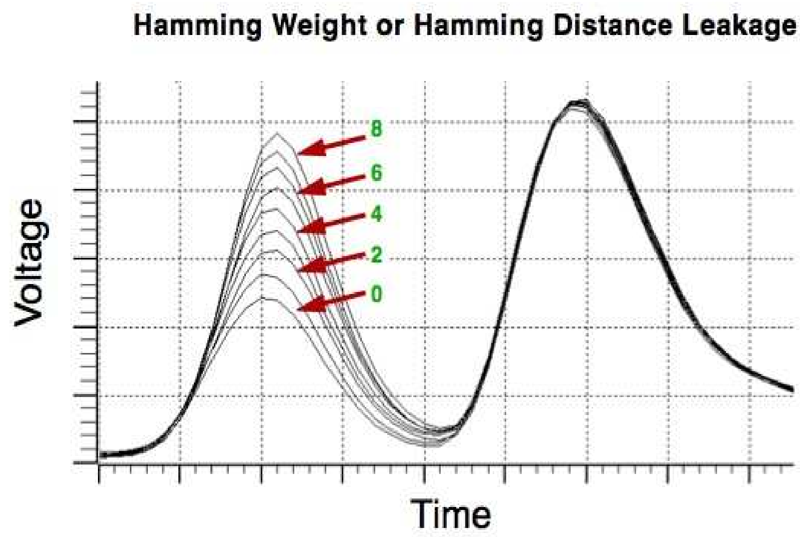
Resistor 20-80 ohm

# More advanced setup for power analysis



# Direct power analysis

```
bool bSimilar = TRUE;
for (short i=0; i<passLength;i++) {
    if (array1[i] != array2[i])
        bSimilar = FALSE;
}
```

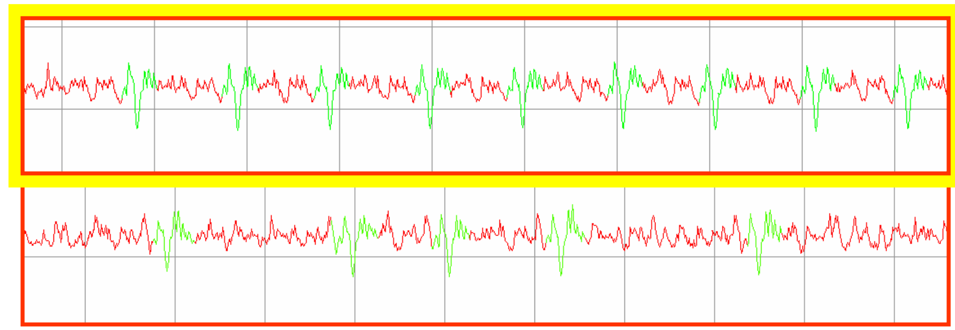



```
getfield_a_this 20;
sload_3;
baload;
getfield_a_this 21;
sload_3;
baload;
if_scmpeq L4;
```

| | P | A | S | S | W | O | R | D |
|---|---|---|---|---|---|---|---|---|
| array1 | P | A | S | S | W | O | R | D |
| array2 | A | B | C | D | E | F | G | H |

# Different sequence of instructions

```
bool bSimilar = TRUE;
for (short i=0; i<passLength;i++) {
    if (array1[i] != array2[i])
        bSimilar = FALSE;
}
```

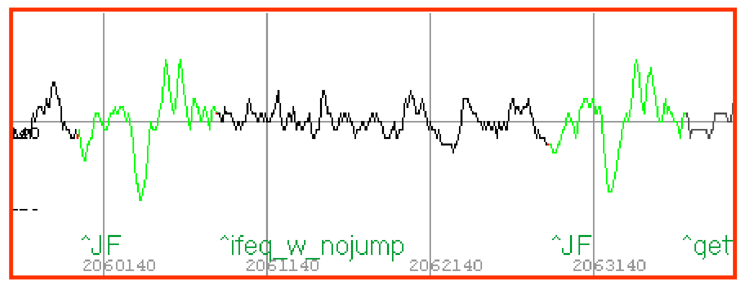


| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | ✓ | ✓ | ✗ | ✗ | ✗ | ✗ | ✓ | ✗ |
| **array1** | **P** | **A** | **S** | **S** | **W** | **O** | **R** | **D** |
| | = | = | | | | | = | |
| **array2** | **P** | **A** | **A** | **B** | **C** | **D** | **R** | **F** |

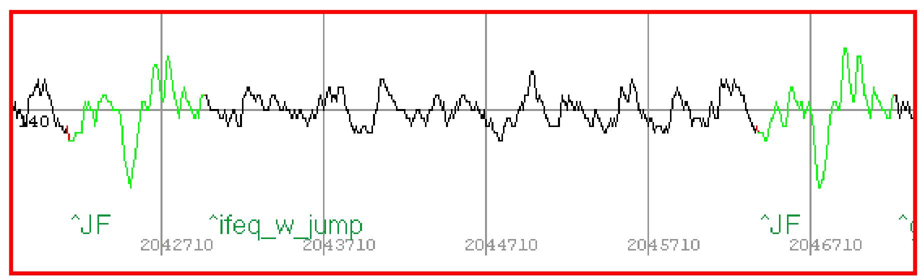
# Sensitive data leakages - Type III.

- Single instruction execution differs
  - depending on the data manipulated
  - e.g., when jump was executed (or not)
- Probably caused by different “microinstructions” for same bytecode instruction (JVM)

***jump not executed (THEN branch)***

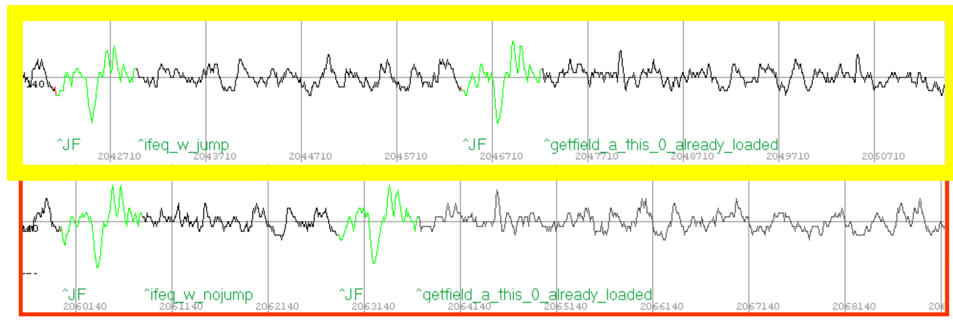



***jump executed (ELSE branch)***



# Different instruction appearance

```
bool bSimilar = TRUE;
bool bFake = TRUE;
for (short i=0; i<passLength;i++) {
    if (array1[i] != array2[i])
        bSimilar = FALSE;
    else
        bFake = FALSE;
}
```

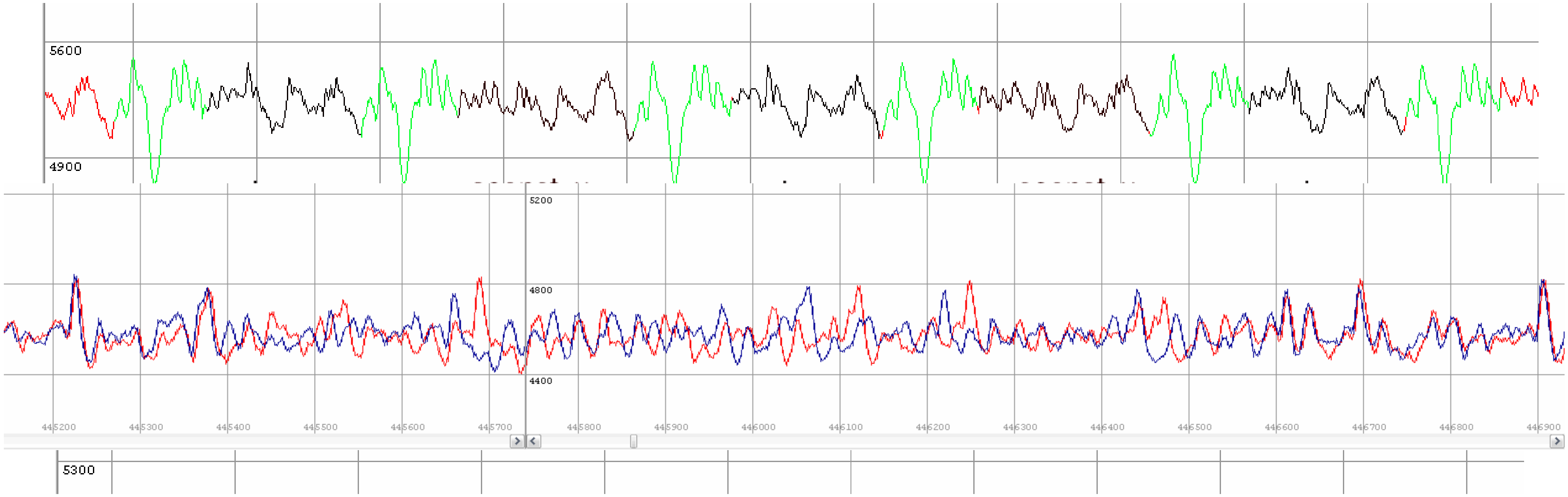



```
getfield_a_this 20;
sload_3;
baload;
getfield_a_this 21;
sload_3;
baload;
if_scmpeq L4;
```

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | ✓ | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| array1 | P | A | S | S | W | O | R | D |
| | = | = | | | | | | |
| array2 | P | A | A | B | C | D | E | F |

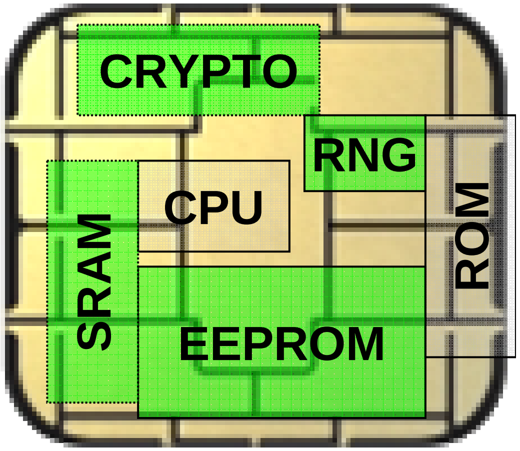
# Situation with current smart cards

- Tested 10 different cards from 4 manufactures
  - 3 with clearly visible bytecode and separators



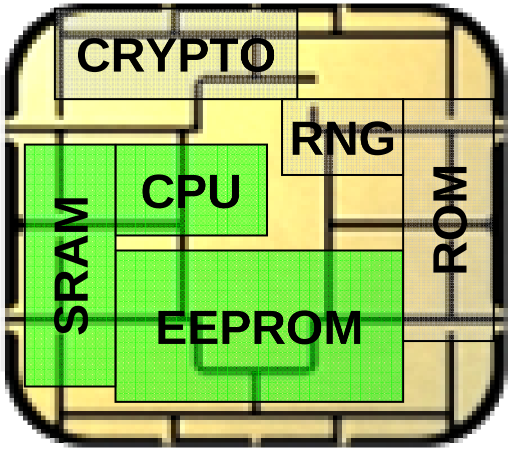
- Caused by used type of the main processor

# Protections on hardware level




- Changes on hardware level
  - masking, randomization, dual-rail logics
  - security vs. speed/memory/chip area
- Disadvantages
  - focused mostly on data protection (algorithm is known)
  - focused mostly on cryptographic coprocessor
  - hard to protect general code executed on JavaCard level
- Expensive and non-flexible solution for customer
  - hardware replacement required (price, logistics)

# Protections on software level




- Changes on software level
  - best practices & secure coding patterns
  - more flexible, can react on actual threats
- Disadvantages
  - limited by underlying hardware
  - may obscure original code functionality
  - additional logical bugs, harder to audit
  - problem with code expandability and maintenance
  - high requirements on developers
- Sometimes the only possibility for a customer

# Co s tim?

# Automated code transformation

# Automated replacement frameworks

*Vulnerable constructions*

*Non-vulnerable constructions*

*Replacement rules*

*Original source code*

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

**???**

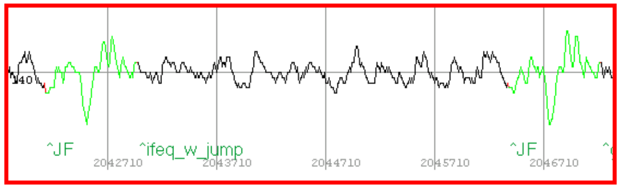
*Transformed source code*

```
bool expr_res = key == 0;
bool expr_res_neg = !(key == 0);
switch (rand_bit) {
  case -1: break;
  case 1: if (expr_res) m_ram1[0] = 1;
          else m_ram1[0] = 0; break;
  case 2: if (expr_res_neg) m_ram1[0] = 0;
          else m_ram1[0] = 1; break;
}
```

## 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

# IfSwitch transformation – naive


```
if (key==0) m_raml[0] = 5;
else m_raml[0] = 7;
```

```
switch ((key == 0) ? 0 : 1) {
  case -1 : throw new Exception(); break;  // never taken
  case 0 : m_raml[0] = 5; break;            // then branch
  case 1 : m_raml[0] = 7; break;            // else branch
}
```

- Original conditional jump still present!

## IfSwitch transformation – r

```
if (key==0) m_raml[0] = 5;
else m_raml[0] = 7;
```

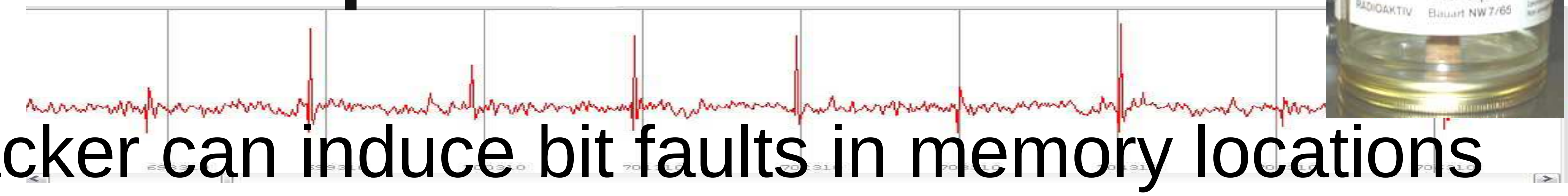
Assumption: comparison and assignment is not leaking

```
boolean expr_res13 = key == 0;
boolean expr_res_neg13 = !(key
switch (__getRandomBit()) {
  case -1: throw new Exception(); break; // never taken
  case 0: if (expr_res13) m_raml[0] = 5;
          else m_raml[0] = 7; break;
  case 1: if (expr_res_neg13) m_raml[0] = 7;
          else m_raml[0] = 5; break;
}
```

Random branch will be taken

- IF THEN ELSE still present, but randomized
  - attacker can distinguish *then* and *else* branch
  - but not *case 0:* and *case 1:* branch

# Another example – 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**

*a_inv* **10100101**

*if (a != ~a_inv) Exception();*
*a = 0x55;*
*a_inv = ~0x55;*

**01010000**

**10101010**

*if (a != ~a_inv) Exception();*
*a = 0x13;*

- Robust protection, but cumbersome for developer

# FaultResistantVariable transformation

```
short i=1, j=1;
if (i==1)
    i+=j;
```

```
private short fault_resistant_short[] = new short[2];
...
short i=__set_short(1,0), j=__set_short(1,1);
if (__get_short(i,0)==1)
    i=__set_short(
        __get_short(i,0)
            +
        __get_short(j,1),
    0);
...
private short __get_short(short value, short id){
    if (fault_resistant_short[id] != value ^ ((1<<15)-1))
        ISOException.throwIt(ISO7816.SW_DATA_INVALID);
    return value;
}
private short __set_short(short value, short id){
    fault_resistant_short[id] = value ^ ((1<<15)-1);
    return value;
}
```

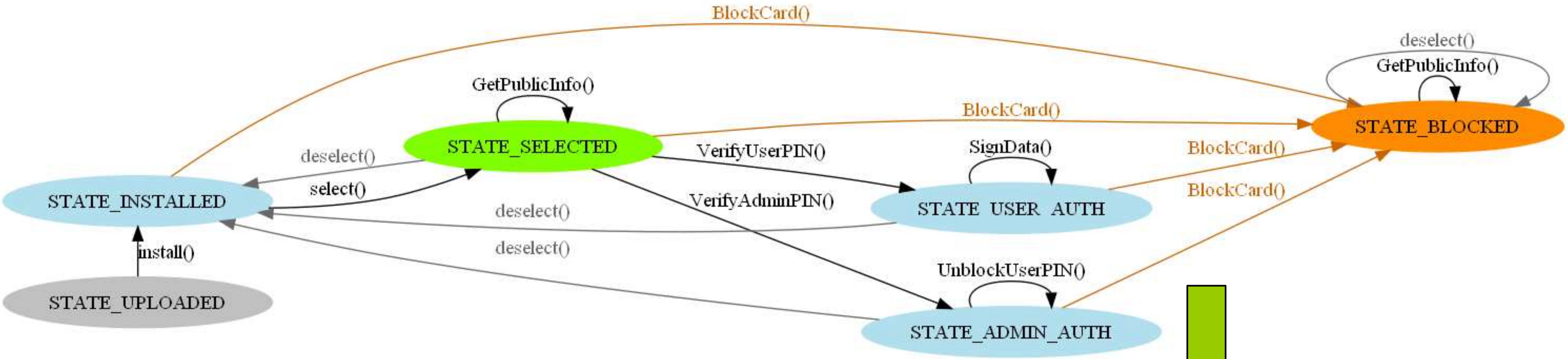
# Applet state transition enforcement

- 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 */;
```

# What can you get right now?

- 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

# Limitations of approach

1. Hidden vulnerabilities still might exist
   - no formal proof
2. Readability of transformed code impacted
   - highly dependent on transformation
3. Possible computation and memory overhead
   - JOpenPGPCard – 2.2x size increase, some speed ↓

# (Near-)future work

- Utilize power of Java code annotations
  - sensitive code/data for targeted replacement
- Security protocols abstractions
  - e.g., secure channel protocol, key diversification…
- Transformations for existing best practices
- Support for other platforms (MS .NET)
- Unit testing for parts of the transformed code
- DOT model as input for formal verification
- …

# Conclusions

- Going from Java to Java Card is easy
- Making efficient Java Card applet is harder
- Making secure Java Card applet is hard
  - platform security, active attacks, logical&coding errors
- CesTa tool is powerful and easy to use!
  - practically usable transformations already available
  - make applet once and with clean logic

***http://CesTa.sourceforge.net***

Thank you for your attention!

Questions

***http://cesta.sourceforge.net***

# References

- ANother Tool for Language Recognition (ANTLR**)**
  - http://www.antlr.org/
- Parr, Terence, *The Definitive ANTLR Reference: Building Domain-Specific Languages* (Raleigh: The Pragmatic Bookshelf, 2007).
- Our parser is derived from publicly available Java 1.5 parser from HABELITZ Software Developments
  - http://antlr.org/grammar/1207932239307/Java1_5Grammars
- StringTemplate engine
  - http://www.stringtemplate.org/