

The JAVA CARD Transaction Mechanism: Specification, Experimentation & Formalisation

Engelbert Hubbers, Wojciech Mostowski, Erik Poll
{hubbers,woj,erikpoll}@cs.ru.nl

Radboud University Nijmegen

Motivation

- ▶ Formal specification and verification of JavaCard for highest level of Common Criteria evaluation
- ▶ incl. what exactly happens when smartcard is subjected to **card tears** (unexpected power loss)?
Card tears more frequent for contactless smartcards than normal smartcards.
- ▶ Can we use this **fault injection** to break security?
- ▶ Can we **prove** this does not break security?

Background: special Java Card features

Smart card memory

- ▶ ROM
- ▶ RAM aka transient: scratchpad memory
- ▶ EEPROM aka persistent: smartcard's harddisk

JavaCard transaction mechanism

- ▶ Several updates to persistent memory can be made into **atomic transaction** with `begin/commitTransaction`
- ▶ A partially completed transaction will be rolled back when **card tear** happens (or when `abortTransaction` is called)

Example transaction

Transactions can be used to keep persistent data (in EEPROM) consistent, eg.

```
JCSystem.beginTransaction();  
    logSize++;  
    log[logSize] = balance;  
    balance = balance+amount;  
JCSystem.endTransaction();
```

- ▶ Log and balance updated in one atomic transaction
- ▶ On card tear during transaction: changes are rolled back, keeping log consistent
- ▶ Consistency properties of persistent data should never be broken, not even temporarily, except during transactions.

Transactions as security vulnerability

Transactions are meant as security countermeasure, but can also introduce new vulnerabilities ...

```
public class AttackerApplet{
    JCSystem.beginTransaction();
    if(!anotherapplet.pin.check("1234",...);)
        { // our guess was incorrect
            while(true){ }; // program hangs: card tear!
        }
    ... // our guess was correct
```

Card tear will roll back the **PIN try counter**, if our guess "1234" was wrong.

Changes to PIN try counter should **never** be rolled back!

By-passing the transaction mechanism

JAVA CARD API offers **non-atomic methods** to by-pass transaction mechanism, but only for assignments in arrays

```
static native short arrayCopyNonAtomic(  
    byte[] src, short srcOff,  
    byte[] dest, short destOff,  
    short length);
```

*“This method **does not use the transaction facility** during the copy operation even if a transaction is in progress. Thus, this method is suitable for use only when the contents of the destination array can be left in a **partially modified state** in the event of a power loss in the middle of the copy operation.”*

Note: arrayCopyNonAtomic has to be native!

Implementing PIN objects

The JavaCard API class `OwnerPIN` provides PIN objects, which include a PIN code and a try counter.

- ▶ Sun's old reference implementation (v2.0 and earlier) can be attacked as discussed earlier, ie. by rolling back the PIN try counter.
- ▶ The new reference implementation (v2.1 and later) uses `arrayCopyNonAtomic`, to prevent this attack.

Can we prove this?

This requires **complete formalisation of the transaction mechanism and non-atomic methods.**

Experimental semantics

SUN's official language specification is not always clear and unambiguous ...

Test applet

Runs over 200 test combinations on the transaction mechanism and non-atomic methods:

- ▶ `arrayCopyNonAtomic` & `arrayFillNonAtomic`
- ▶ persistent & transient arrays
- ▶ inside & outside of a transaction
- ▶ abort by card tear or `abortTransaction`
- ▶ card tear during non-atomic method
- ▶ ...

What does the SUN spec **not** say? (1)

What if... an array is updated with a regular conditional update and with a non-atomic method within the same transaction? Eg.

```
a[0] = 0;
b[0] = 2;
beginTransaction();
    a[0] = 1; // conditional update
    arrayCopyNonAtomic(b,0,a,0,1); // a[0] = 2;
abortTransaction(); // a[0] rolled back to 0 or 2?
```

This is bizarre code, but language specification should define the semantics of bizarre code, too.

Experiments with real cards show that they all roll back to the value just before the first conditional update (here 0).

Formalisation

Clarified “non-tear” semantics

Possible to formalise behaviour of transactions and non-atomic methods in the **absence** of card tears.

KeY system

Verification for JAVA CARD programs based on Dynamic Logic – incl. with semantics of transactions and non-atomic methods.

Case studies

- ▶ Verified new reference implementation of **OwnerPIN** ie. verified PIN try counter decreased irrespective of any ongoing transaction
- ▶ Verification of old reference implementation failed, as it should.

What does the SUN spec **not** say? (2)

The spec says that card tear *during* non-atomic method leaves array in '**partially modified**' state. What does this mean?

Depends on the card! It can be

1. mixture of old and new values, as you'd expect
2. all old or all new values, ie. *better* than you'd expect
(NB `arrayCopyNonAtomic` is atomic on these cards! The method's name is misleading, the crucial property is not that it is non-atomic, but that it is *unrollbackable*.)
3. the array can be zeroed out
4. apparently random values, but always the same random values

One could argue that 3 and 4 violate the specification (unless '**partially modified**' means '**totally messed up**').

Revision in SUN spec for Java Card version 2.2

Language spec weakened, by additional note saying that 'partially modified' means 'not predictable'.

*“**Note** – The contents of an array component which is updated using the `arrayCopyNonAtomic` method while a transaction is in progress, is **not predictable**, following the abortion of the transaction.”*

This does not solve the problem, because many cards are also not predictable if this happens while **no** transaction is in progress.

This behaviour of `arrayCopyNonAtomic` is a bug.

Attacking PIN implementations

- ▶ On cards where card tears produce random values (category 4), the improved reference implementation of `OwnerPIN` is not secure:

card tear could set try counter to random value

- ▶ We can demonstrate this attack.
- ▶ We have (so far) not been able to attack actual `OwnerPIN` implementations on these smartcards

These are programmed more defensively, or have native implementations that do not share the bug of `arrayCopyNonAtomic`.

Proving security of PIN implementations

- ▶ We implemented a defensive OwnerPIN, which is secure against fault attacks by card tears, both for the Sun spec and for every card we know that breaks this spec.
- ▶ Security has been proved using **model checking** (Uppaal):
 1. by hand building a model of the code which includes all possible behaviour under card tears, incl.
 - ▶ 'abort' after every bytecode instruction
 - ▶ 'random' assignment by `arrayCopyNonAtomic` calls
 2. then letting model-checker check all possible executions
- ▶ This technique could be used to include other fault injections.

Results

- ▶ Complete understanding of the transaction mechanism, both of what the spec says and what cards actually do.
- ▶ Extensive test suite for testing transaction mechanism
- ▶ Complete formalisation, incl. transactions and non-atomic methods, in KeY tool.
Enables **program verification**, incl.
 - ▶ functional correctness if no card tear happens
 - ▶ preservation of 'consistency' if card tears happen
 - ▶ *but not*: correctness under 'faults' by card tears
- ▶ Last property can be checked for crucial components, for any card behaviour we are aware of, by **model checking**.
Ongoing work: including other faults, ie. *pre-silicon fault injection analysis*

Conclusions

- ▶ Transaction mechanism is **tricky**
 - ▶ for developers of cards, and
 - ▶ for developers of applications for these cards.
- ▶ Many smartcards still fail to meet the official Sun spec, despite weakening of the spec in version 2.2.
- ▶ Programmers have to be very cautious using non-atomic methods to by-pass transaction mechanism, and possibly check card-specific behaviour.
- ▶ Sensible programming guideline: **'do not use non-atomic methods'** (except for transient data in RAM)
Can be enforced by static source code analysis.