

JAVA CARD Non-Atomic Methods

Wojciech Mostowski

woj@cs.ru.nl

Radboud University Nijmegen

Outline

JAVA CARD transaction mechanism

Non-Atomic methods

Experiments

JAVA CARD DL formalisation

Examples

Wrap-up

JAVA CARD – “subset” of JAVA

Not really. . .

- ▶ Memory model: ROM, RAM, EEPROM
- ▶ Transaction mechanism
- ▶ Applet firewall

Accurate reasoning about JAVA CARD applets is **more difficult** than first thought.

JAVA CARD – “subset” of JAVA

Not really. . .

- ▶ Memory model: ROM, RAM, EEPROM
- ▶ Transaction mechanism
- ▶ Applet firewall

Accurate reasoning about JAVA CARD applets is **more difficult** than first thought.

Transaction Mechanism

Memory

- ▶ ROM – read only
- ▶ RAM – read/write transient
- ▶ EEPROM – read/write persistent – **storage memory**

Transaction Mechanism

- ▶ To ensure consistency of the **storage memory**
- ▶ Updates to persistent memory locations can be gathered in **atomic blocks** with `beginTransaction` and `commitTransaction`
- ▶ A transaction can be aborted with `abortTransaction` – update roll-back
- ▶ transient memory **does not** participate in transactions

Transaction Mechanism

Memory

- ▶ ROM – read only
- ▶ RAM – read/write transient
- ▶ EEPROM – read/write persistent – **storage memory**

Transaction Mechanism

- ▶ To ensure consistency of the **storage memory**
- ▶ Updates to persistent memory locations can be gathered in **atomic blocks** with `beginTransaction` and `commitTransaction`
- ▶ A transaction can be aborted with `abortTransaction` – update roll-back
- ▶ transient memory **does not** participate in transactions

Bypassing the Transaction Mechanism

What if...

one also wants to exclude (parts of) persistent memory from the transaction?

PIN try counter

Persistent, yet should not be rolled back upon transaction abort – security breach.

For this purpose JAVA CARD API offers two **non-atomic methods**.

Bypassing the Transaction Mechanism

What if...

one also wants to exclude (parts of) persistent memory from the transaction?

PIN try counter

Persistent, yet should not be rolled back upon transaction abort – security breach.

For this purpose JAVA CARD API offers two **non-atomic methods**.

Bypassing the Transaction Mechanism

What if...

one also wants to exclude (parts of) persistent memory from the transaction?

PIN try counter

Persistent, yet should not be rolled back upon transaction abort – security breach.

For this purpose JAVA CARD API offers two **non-atomic methods**.

Non-Atomic Methods

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

```
static native short arrayFillNonAtomic(  
    byte[] bArray, short offset, short length, byte value);
```

“This method does not use the transaction facility during the copy/fill 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.”

Limitations

Only byte array elements can be excluded from a transaction

Coding is not straightforward:

```
byte tryCounter;
```

has to be coded as:

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

All updates to such arrays have to be performed with non-atomic methods:

```
// decrease try counter:  
temps[0] = (byte)(tryCounter[0]-1);  
Util.arrayCopyNonAtomic(  
    temps, (short)0, tryCounter, (short)0, (short)1);
```

Limitations

Only byte array elements can be excluded from a transaction

Coding is not straightforward:

```
byte tryCounter;
```

has to be coded as:

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

All updates to such arrays have to be performed with non-atomic methods:

```
// decrease try counter:  
temps[0] = (byte)(tryCounter[0]-1);  
Util.arrayCopyNonAtomic(  
    temps, (short)0, tryCounter, (short)0, (short)1);
```

What Else Does the SUN Spec Say?

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

Meaning...

A memory location (e.g. PIN try counter) can be left with a **random** value upon transaction abort. And it is **OK by the spec!**

Meaning...

By the spec, non-atomic methods are useless for transaction bypassing. **Reference implementation** of the `OwnerPIN` is in principle unsafe/broken (successful attack).

So...

Tests, tests, tests, to see what cards really do.

What Else Does the SUN Spec Say?

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

Meaning. . .

A memory location (e.g. PIN try counter) can be left with a **random** value upon transaction abort. And it is **OK by the spec!**

Meaning. . .

By the spec, non-atomic methods are useless for transaction bypassing. [Reference implementation](#) of the `OwnerPIN` is in principle unsafe/broken (successful attack).

So. . .

Tests, tests, tests, to see what cards really do.

What Else Does the SUN Spec Say?

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

Meaning. . .

A memory location (e.g. PIN try counter) can be left with a **random** value upon transaction abort. And it is **OK by the spec!**

Meaning. . .

By the spec, non-atomic methods are useless for transaction bypassing. [Reference implementation](#) of the `OwnerPIN` is in principle unsafe/broken (successful attack).

So. . .

Tests, tests, tests, to see what cards really do.

What Else Does the SUN Spec Say?

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

Meaning. . .

A memory location (e.g. PIN try counter) can be left with a **random** value upon transaction abort. And it is **OK by the spec!**

Meaning. . .

By the spec, non-atomic methods are useless for transaction bypassing. [Reference implementation](#) of the `OwnerPIN` is in principle unsafe/broken (successful attack).

So. . .

Tests, tests, tests, to see what cards really do.

What Does the SUN Spec Not Say?

What if...

A memory cell/array element is updated with a regular statement (atomically) and with a non-atomic method within the same transaction?

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

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

Data is backed-up just before it is updated conditionally for the first time. Although this is logical, it is not in the SUN spec!

What Does the SUN Spec Not Say?

What if...

A memory cell/array element is updated with a regular statement (atomically) and with a non-atomic method within the same transaction?

<pre>a[0] = 0; beginTransaction(); a[0] = 1; arrayFillNonAtomic(a,0,1,2); // a[0] = 2; abortTransaction(); // a[0] == 0;</pre>	<pre>a[0] = 0; beginTransaction(); arrayFillNonAtomic(a,0,1,2); // a[0] = 2; a[0] = 1; abortTransaction(); // a[0] == ?;</pre>
---	---

Data is backed-up just before it is updated conditionally for the first time. Although this is logical, it is not in the SUN spec!

What Does the SUN Spec Not Say?

What if...

A memory cell/array element is updated with a regular statement (atomically) and with a non-atomic method within the same transaction?

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

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

Data is backed-up just before it is updated conditionally for the first time. Although this is logical, it is not in the SUN spec!

What Does the SUN Spec Not Say?

What if...

A memory cell/array element is updated with a regular statement (atomically) and with a non-atomic method within the same transaction?

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

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

Data is backed-up just before it is updated conditionally for the first time. Although this is **logical**, it is **not** in the SUN spec!

Experiments

Test applet (Nijmegen gang & myself)

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

Experiments

Results

- ▶ some cards take the “full liberty” of the SUN spec – “random” values in the test array upon card tear (but not upon the programmatic abort).
- ▶ “random” = random, but usually the same set of random values (most likely part of the card’s memory footprint)
- ▶ some cards do the “right thing” – well defined, deterministic behaviour – no random values
- ▶ some cards do something in between – the test array is zeroed out – special case of random
- ▶ almost none of the cards are fully SUN spec compliant – card tear that occurs during a non-atomic method call **outside** of a transaction gives random or zeroed test array.

This is not allowed by SUN spec! (Unless “partially modified” means “totally messed up”)

Experiments

Results

- ▶ some cards take the “full liberty” of the SUN spec – “random” values in the test array upon card tear (but not upon the programmatic abort).
- ▶ “random” = random, but usually the same set of random values (most likely part of the card’s memory footprint)
- ▶ some cards do the “right thing” – well defined, deterministic behaviour – no random values
- ▶ some cards do something in between – the test array is zeroed out – special case of random
- ▶ almost none of the cards are fully SUN spec compliant – card tear that occurs during a non-atomic method call **outside** of a transaction gives random or zeroed test array.

This is not allowed by SUN spec! (Unless “partially modified” means “totally messed up”)

Experiments

Transactions & non-atomic methods semantics

Based on the test results for the well-behaved cards, the holes in the specification have been filled in – **experimental semantics**

Meaning...

For the KeY formalisation an “idealised” (sensible & deterministic) semantics has been used, that only a handful of cards actually implement.

Experiments

Transactions & non-atomic methods semantics

Based on the test results for the well-behaved cards, the holes in the specification have been filled in – **experimental semantics**

Meaning. . .

For the KeY formalisation an “idealised” (sensible & deterministic) semantics has been used, that only a handful of cards actually implement.

Formalisation in KeY

Basic transaction handling

For each transaction split the proof into two branches (commit & abort), on the abort branch special (shadowed) updates:

```
this.a = 0;  
beginTransaction();  
this.a = 1; // this.a' = 1;  
abortTransaction();  
// this.a == 0
```

Formalisation in KeY

Transaction suspending

In JAVA CARD DL model now there are two more transaction statements: `jvmSuspendTransaction`, `jvmResumeTransaction`.

Transaction suspended – updates to persistent data unconditional (non-shadowed), e.g. inside a non-atomic method.

```
public static short arrayFillNonAtomic(  
    byte[] bArray, short offset, short len, byte val) {  
    jvmSuspendTransaction();  
    for(short i = 0; i < len; i++)  
        bArray[(short)(offset + i)] = val;  
    jvmResumeTransaction();  
    return (short)(offset + len);  
}
```

Formalisation in KeY

Transaction suspending

In JAVA CARD DL model now there are two more transaction statements: `jvmSuspendTransaction`, `jvmResumeTransaction`.

Transaction suspended – updates to persistent data unconditional (non-shadowed), e.g. inside a non-atomic method.

```
public static short arrayFillNonAtomic(  
    byte[] bArray, short offset, short len, byte val) {  
    jvmSuspendTransaction();  
    for(short i = 0; i < len; i++)  
        bArray[(short)(offset + i)] = val;  
    jvmResumeTransaction();  
    return (short)(offset + len);  
}
```

Formalisation in KeY

Notice...

- ▶ Committing and restarting the transaction instead of suspending is (obviously) not a good solution
- ▶ Suspend/resume statements are only interesting in the scope of a transaction about to abort. Otherwise suspend/resume can be safely ignored

Formalisation in KeY

Three states of a transaction in JAVA CARD DL

- ▶ transaction started and about to commit
- ▶ transaction started and about to abort
- ▶ transaction suspended (can only result from the abort state)

Now each modality has four versions: regular, `_trc`, `_tra`, `_susp`

Keeping track of array updates

Need to know whether an array element `i` has been conditionally updated inside a transaction – new implicit field (array):

```
a.<trInit>[i] := FALSE
```

JAVA CARD DL Rules

Modified:

- ▶ Assignment rules in the scope of the `_tra` modalities
- ▶ Update the information on what elements of an array have been conditionally assigned

Added:

- ▶ Assignment rules in the scope of the `_susp` modalities
- ▶ Check if an array element has been conditionally assigned, and resp. do a conditional or unconditional assignment
- ▶ For the throughout modality check the strong invariant

Examples

```
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();
```

```
// a.<trInit>[*] == false
a[0] = 0;
beginTransaction();
  a.<trInit>[0] = true; a[0]' = 1;
  if(a.<trInit>[0]) a[0]' = 2;
  else a[0] = 2;
abortTransaction();
// a[0] == 0;
```

```
// a.<trInit>[*] == false
a[0] = 0;
beginTransaction();
  if(a.<trInit>[0]) a[0]' = 2;
  else a[0] = 2;
  a.<trInit>[0] = true; a[0]' = 1;
abortTransaction();
// a[0] == 2;
```

Examples

```
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();
```

```
// a.<trInit>[*] == false
a[0] = 0;
beginTransaction();
  a.<trInit>[0] = true; a[0]' = 1;
  if(a.<trInit>[0]) a[0]' = 2;
  else a[0] = 2;
abortTransaction();
// a[0] == 0;
```

```
// a.<trInit>[*] == false
a[0] = 0;
beginTransaction();
  if(a.<trInit>[0]) a[0]' = 2;
  else a[0] = 2;
  a.<trInit>[0] = true; a[0]' = 1;
abortTransaction();
// a[0] == 2;
```

Optimising JAVA CARD DL

```
jvmSuspendTransaction();  
for(short i = 0; i < len; i++)  
    bArray[(short)(offset + i)] = val;  
jvmResumeTransaction();
```

becomes:

```
// check if the arguments are well defined,  
// throw exceptions if necessary  
jvmArrayFillNonAtomic(bArray, offset, len, val);
```

- ▶ Dedicated taclet for `jvmArrayFillNonAtomic` – quantified update
- ▶ Native methods – **native taclets** to accurately reflect JAVA CARD VM behaviour
- ▶ Taclet options – possible to switch the whole transaction mechanism and/or strong invariants off

Example: OwnerPIN Reference Implementation

```
JCSystem._transactionDepth = 0 & !pin = null &
!pin._triesLeft = null &
...rest of the OwnerPIN class invariant
_triesLeft@pre = pin._triesLeft[0] & result = -1 →
{ try {
    JCSystem.beginTransaction();
    if(pin.check(pin,offset,len)) result = 1;
    else result = 0;
    if(b) JCSystem.abortTransaction();
    else JCSystem.commitTransaction();
} catch(Exception ex) {})(
    (_triesLeft@pre = 0 →
        result = 0 & pin._triesLeft[0] = 0) &
    (_triesLeft@pre > 0 → (result = 0 →
        pin._triesLeft[0] = _triesLeft@pre - 1)))
```

Conclusions/Discussion Points

- ▶ Transactions + non-atomic methods formalised & implemented
- ▶ Solution **generic** – other semantics of transactions possible (within “certain” limits)
- ▶ Non-trivial examples verified
- ▶ Real cards reveal problems – cheap fault injection attacks
- ▶ A fool-proof implementation of OwnerPIN quite involved – correctness subject to KeY verification and model checking (both verified!)
- ▶ Should implementation bugs/features be modelled/formalised in KeY – taclet options for each card manufacturer?
- ▶ SUN specification should be “straightened out”

Future Work

Is that all regarding the formalisation of **complications-free** language JAVA CARD?

Not yet:

- ▶ Reference resetting (for newly created objects) upon transaction abort – piece of cake with new object creation schema?
- ▶ Applet firewall – JAVA CARD SecurityException

More:

- ▶ JAVA CARD keywords in JML: `cardtear`,
`\conditionally_assigned`, `\backup/\shadow`, etc.

Future Work

Is that all regarding the formalisation of **complications-free** language JAVA CARD?

Not yet:

- ▶ Reference resetting (for newly created objects) upon transaction abort – piece of cake with new object creation schema?
- ▶ Applet firewall – JAVA CARD `SecurityException`

More:

- ▶ JAVA CARD keywords in JML: `cardtear`,
`\conditionally_assigned`, `\backup/\shadow`, etc.

Future Work

Is that all regarding the formalisation of **complications-free** language JAVA CARD?

Not yet:

- ▶ Reference resetting (for newly created objects) upon transaction abort – piece of cake with new object creation schema?
- ▶ Applet firewall – JAVA CARD SecurityException

More:

- ▶ JAVA CARD keywords in JML: `cardtear`,
`\conditionally_assigned`, `\backup/\shadow`, etc.