Transactions and non-atomic API methods in Java Card: specification ambiguity and strange implementation behaviours

Engelbert Hubbers and Erik Poll
SoS Group, NIII, Faculty of Science, Radboud University Nijmegen
{hubbers,erikpoll}@cs.ru.nl

Abstract. This paper discusses an ambiguity in Sun's specification of the Java Card™ platform, which we noticed in the course of developing the precise formal description of the Java Card transaction mechanism presented in [HP03]. The ambiguity concerns the Java Card transaction mechanism, more in particular the interaction of the transaction mechanism and two Java Card API methods, the methods `arrayCopyNonAtomic` and `arrayFillNonAtomic` in the class `javacard.framework.Util`.

The paper also describes the experiments we performed with smartcards of different manufacturers (IBM and Schlumberger) to find out the behaviour actually implemented on these card. Interestingly, these experiments revealed some unexpected (and unexplainable) behaviour of these two methods on some cards.

1 Introduction

The Java Card programming language for smartcards is usually presented as a subset of Java. However, Java Card has several features not present in standard Java, which are specific to smartcards. One of these features is the distinction between persistent memory (EEPROM or Flash) and transient memory (RAM): data stored in transient memory is lost as soon as the card loses power, whereas data stored in persistent memory is preserved. Another of these features is the transaction mechanism, which is provided to cope with the possibility of so-called card tears, i.e. the sudden loss of power that occurs when a smartcard is torn from the reader. The transaction mechanism can be used to ensure that transactions will be rolled back in the event they are interrupted by a card tear.

In the course of developing the formal semantics of the transaction mechanism presented in [HP03], we noticed a potential ambiguity in Sun's API specification [Jav99]. This Java Card API specification states for two methods in the `javacard.framework.Util` class, the methods `arrayCopyNonAtomic` and `arrayFillNonAtomic`, that they “do not use the transaction mechanism”, in that the effect of these methods will not be roll-backed in the event of a card tear. However, as we explain in Section 2.2, this leaves room for interpretation in case the methods are combined with methods that do use the transaction mechanism. Of course, finding such ambiguities is an important point of developing formal descriptions.

We should note that this ambiguity only occurs in some rather contrived examples, which are highly unlikely to occur in ‘normal’ Java Card programs. Still, the language specification should be unambiguous for any legal Java Card program, especially since malevolent programmers are likely to try out contrived examples of code in an attempt to by-pass or break security of the platform.

The ambiguity is essentially an instance of “feature interaction”: it is the interaction of the transaction mechanism with the methods `arrayCopyNonAtomic` and `arrayFillNonAtomic` that is not unambiguously specified. The transaction mechanism is tricky to specify precisely because it interacts with other API methods. The fact that the transaction mechanism is fundamentally hard to specify is witnessed by the fact that even in the most recent release of the Java Card specification, version 2.2, the specification of the API class which interacts with the transaction mechanism, `javacard.framework.PIN`, has been improved.

To resolve the ambiguity in the Java Card we carried out experiments on smartcard to see how these implement the Java Card standard. These experiments demonstrate that we can observe...
differences between implementations of the transaction mechanism on different smartcards, IBM JCOP and SchlumbergerSema Palmera cards. More interesting is that the experiments demonstrate some very strange behaviour, which we cannot explain. We have not been able to exploit this strange behaviour to by-pass or attack platform security, though.

The overall structure of the paper is as follows: Section 2 briefly explains the Java Card transaction mechanism, Then 2.2 explains the ambiguity in the specification of the methods arrayCopyNonAtomic and arrayFillNonAtomic, and Section 3 gives the details of the experiments we carried out on physical smartcards.

2 Transactions in Java Card

This section briefly explains the transaction mechanism of Java Card. For a more complete explanation, see [Che00] or the Java Card Runtime Environment (JCRE) specification [Sun00] and the Java Card API specification [Jav99].

2.1 The Java Card transaction mechanism

Java Card provides a distinction between persistent memory (EEPROM or Flash) and transient memory (RAM). A smartcard does not have its own power supply, but relies on the card reader for its power supply. This means that whenever a smartcard is removed from the reader, all data stored in transient memory (RAM) is lost, and only data stored in persistent memory is preserved. By default, in Java Card all objects and their fields are allocated in persistent memory, and transient data is only used for the stack and for specially designated fields that serve as ‘scratch pad’ memory. The special scratch pad fields are always arrays, and are called transient arrays. NB it is the contents of such a transient array that is transient, the reference to such a transient array is (typically) persistent.

In many card readers it is possible to tear the smartcard out of the reader while it is in operation. Such a so-called card tear results in a sudden loss of power. Clearly, a card tear occurring in the middle of some operation, could cause problems and possibly leave the smartcard in an inconsistent state. E.g., a card tear during the debit operation on an electronic purse could result in electronic money disappearing or being created.

To cope with card tears, the Java Card API offers a so-called transaction mechanism. This can be used to ensure that several updates to persistent memory are executed as a single atomic operation, i.e. either all updates are performed or none at all. The Java Card API offers three methods for this: beginTransaction, commitTransaction and abortTransaction. After a beginTransaction all changes to persistent data are executed conditionally. To quote the JCRE specification [Sun00, Section 7.5]:

“If power is lost (tear) or the card is reset or some other system failure occurs while a transaction is in progress, then the JCRE shall restore to their previous values all [persistent] fields and array components conditionally updated since the previous call to JCSystem.beginTransaction.”

Note that changes to transient data, including local variables, are executed unconditionally. The transaction is ended by commitTransaction or abortTransaction; in the former case the updates are committed, in the latter case the updates are discarded. If a card tear occurs during a transaction, any updates to persistent data done during that transaction are discarded.

For example, suppose that u and t are two arrays with the same length, allocated in EEPROM (which is the default). Then the following code fragment

```java
JCSysystem.beginTransaction();
for(int i=0; i<t.length; i++) t[i] = u[i];
JCSysystem.endTransaction();
```
will copy the contents of u to t in a single atomic operation. I.e. if a card tear occurs during execution of the code fragment, then either all array elements will be copied or none will be copied.

The Java Card API in fact provides an atomic array copy method, in the class Util: the invocation

```
Util.arrayCopy(u,0,t,0,t.length);
```

is equivalent to the code fragment above.

So any updates of persistent memory during a transaction are only done conditionally, and all these conditional updates are committed in one atomic action at the very end of the transaction. To implement the transaction mechanism, the Java Card platform performs some special clean-up operations every time the card powers up, to undo the effects of any unfinished transaction. Different techniques to implement this are discussed in [Oes99].

2.2 The methods arrayCopyNonAtomic and arrayFillNonAtomic

The class javacard.framework.Util provides several methods for manipulating byte arrays. Byte arrays are heavily used in Java Card programs; all communication between a Java Card smartcard and the outside world is done using byte arrays. For two of these methods in javacard.framework.Util:

- arrayCopyNonAtomic
- arrayFillNonAtomic

the Java Card API specification [Jav99] says that they “do not use the transaction mechanism”. For arrayCopyNonAtomic the specification says that

> “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.”

The text for the arrayFillNonAtomic is similar. In other words, if a card tear occurs during invocations of these operations, the state of the destination array can be only partially modified, even if the invocation occurs in a transaction. For example, the following code fragment

```
JCSystem.beginTransaction();
Util.arrayCopyNonAtomic(u,0,t,0,t.length);
JCSystem.endTransaction();
```

can leave the array u partially copied when a card tear occurs, despite the fact that the invocation of arrayCopyNonAtomic occurs inside a transaction.

Not surprisingly, both these methods are declared as native; it would be impossible to implement this behaviour in Java Card.

The code fragment below, loosely based on the reference implementation of the Java Card API class OwnerPIN, illustrates why and how one might want to use arrayCopyNonAtomic:

**Example 1.** An OwnerPIN object records a PIN code in pin and records the number of attempts that are left to guess the correct PIN code triesLeft[0].

```java
package javacard.framework;

public class OwnerPIN implements PIN{
    private byte[] pin; // array of length 4 storing the PIN code
    private byte[] triesLeft , temps; // arrays of length 1

    ... 

    private void decrementTriesRemaining(){
        temps[0] = (byte)(triesLeft[0]−1);
        Util.arrayCopyNonAtomic(temps, 0, triesLeft, 0, 1);
    }
}
```
public boolean check(byte[] guess) {
    if (triesLeft[0] == 0) return false;
    decrementTriesRemaining();
    if (Util.arrayCompare(guess, 0, pin, 0, 4) == 0) {
        triesLeft[0] = 3;
        return true;
    } else return false;
}

The method check checks if a supplied PIN code is correct, and resets the value of triesLeft[0] to 3 if it is correct. The invocation of decrementTriesRemaining in check will decrease triesLeft[0]; the way in which this is done, using arrayCopyNonAtomic, guarantees that this reduction of triesLeft[0] cannot be rolled-back, in case check is called within a transaction. Implementing decrementTriesRemaining as

    triesLeft[0] = triesLeft[0] - 1;

might allow an infinite number of guesses of the PIN code in case check was called during a transaction, by an attacker that generates a card tear after each incorrect guess of the PIN code.

(The only reason for using a byte array of length 1 rather than a byte field to record the number of tries that are left is that unconditional updates are only possible using arrayCopyNonAtomic and are therefore only possible for array entries.)

2.3 The problem

The problem with the specification of arrayCopyNonAtomic above is that it is not clear what should happen if in a transaction both one of these non-atomic methods and normal assignments are used to update the same (persistent) field.

Example 2. Consider the following program fragment

```java
for (int i = 0; i < t.length; i++) t[i] = 0;
for (int i = 0; i < t.length; i++) u[i] = 2;
JCSystem.beginTransaction();
for (int i = 0; i < t.length; i++) t[i] = 1;
arrayCopyNonAtomic(u, 0, t, 0, t.length); // assigns 2 to all t[i]
JCSystem.commitTransaction();
```

Here both arrayCopyNonAtomic and normal assignments are used to update the same persistent array t.

What is not clear in this situation is what should happen if a card tear occurs during, say, just before the invocation of commitTransaction, i.e. after completion of the invocation of arrayCopyNonAtomic. There are two possible interpretations of the Java Card specs:

- One could argue that the t[i] should keep their values 2 assigned to them by arrayCopyNonAtomic, because this method “does not use the transaction mechanism”, so the effects of arrayCopyNonAtomic should not be undone.
- On the other hand, one could argue that the t[i] should be reset to their “previous values”, i.e. the value 0 they had upon entering the transaction, because the assignments to them in the for loop do use the transaction mechanism.

In experiments on actual cards we found that t[i] are reset to the value 0 they had upon entering the transaction. More details about these experiments are given in Section 3.

The example above first performs a normal update on t and then uses arrayCopyNonAtomic. the same issue arises if we reverse this:

Example 3. Consider the following program fragment, where again both arrayCopyNonAtomic and normal assignments are used to update the same persistent array t.
What should happen if a card tear occurs during, say, just before the invocation of `commitTransaction`?

Here in experiments on cards it turned out that the $t[i]$ are reset 2. So, on the cards we experimented with, in Example 2 the effects of `arrayCopyNonAtomic` are undone, but in Example 3 the effects of `arrayCopyNonAtomic` are not undone. More details about these experiments are given in Section 3. This suggests that in the following quote from the JCRE specification [Sun00, Section 7.5]

“If power is lost (tear) or the card is reset or some other system failure occurs while a transaction is in progress, then the JCRE shall restore to their previous values all [persistent] fields and array components conditionally updated since the previous call to `JCSystem.beginTransaction`.”

we should read “their previous values” as

“the values they had directly prior to the first conditional update after the previous call to `JCSystem.beginTransaction`.”

where a “conditional update” is any assignment that is not the effect of `arrayCopyNonAtomic` or `arrayFillNonAtomic`.

So, as Example 3 illustrates, we should not read “their previous values” as

“the values they had upon entering the transaction.”

as this interpretation would mean that the $t[i]$ are reset to 0 and not to 2 in Example 3.

Implementing a transaction mechanism involves some shadow bookkeeping: for any persistent data that is altered during a transaction, both the new and the old value have to be recorded; the former is needed in case the transaction is successfully completed, the latter is needed in case of a roll-back. The results of the examples above suggest that in the cards we tested, back-up copies of old values of fields are made directly prior to the first conditional update in the transaction.

We cannot determine whether these cards implement the transaction mechanism using the so-called ‘optimistic’ approach or the ‘pessimistic’ approach as described in [Oes99].

Of course, the code fragments above are very contrived. The specification is only unclear in transactions in which both the non-atomic methods and normal assignments are used to update the same (persistent) field, something one would not expect to happen in normal Java Card code. Still, the language specification should be unambiguous for any legal Java Card program. Malevolent programmers are likely to try out contrived examples of code in an attempt to by-pass or break security of the platform.

3 Experiments

We carried out some experiments with test applets executing on physical smartcards to see how these smartcards behave for the cases like the ones discussed in the previous section. We have tested this applet on three different cards. The IBM JCOP 21id and 31bio cards. And the SchlumbergerSema Palmera card M256LPALP2. The JCOP cards implement Java Card 2.1.1; the Palmera card implements version 2.1. Since the two JCOP cards showed the same behaviour, we will treat them as one type of card from here.
3.1 Generating card tears

Carrying out these experiments is not completely trivial, as it is hard to generate a card tear at exactly the right moment.

A trick we used to be able to generate to card tears at a specific program point was to include non-terminating repetitions in the code. For example, the code fragment

```java
1 JCSystem.beginTransaction();
2 for (i = 0; i < t.length; i++) { t[i]++;
3 while (true) {
4  arrayCopyNonAtomic(u,0,t,t.length);
5  arrayCopyNonAtomic(v,0,t,t.length);
6}
7 JCSystem.commitTransaction();
```

contains an infinite loop in line 3, which allows us to tear out the card after execution of the statements in line 2 and before the `arrayCopyNonAtomic` in line 4. After waiting a few seconds (depending on the length of the array) you can be quite sure that the applet is in the loop.

An alternative would be to replace this non-terminating repetitions by an invocation of `abortTransaction`. This only works since we are only looking at persistent memory. The difference between a card tear and an `abortTransaction` can only be distinguished by looking at the transient memory.

Making sure that a card tear takes place during execution of an invocation of `arrayCopyNonAtomic` is more difficult. For this one cannot use the tricks mentioned above. One possibility would be to use special hardware, such as a card reader which can produce a card tear at a precise time, and count CPU cycles or observe power consumption (as in DPA attacks) to see when copying starts. Instead, we used the following trick of putting calls to `arrayCopyNonAtomic` inside an infinite loop, as in Example 5.

```java
1 JCSystem.beginTransaction();
2 for (i = 0; i < t.length; i++) { t[i]++;
3 while (true) {
4  arrayCopyNonAtomic(u,0,t,t.length);
5  arrayCopyNonAtomic(v,0,t,t.length);
6  arrayCopyNonAtomic(u,0,t,t.length);
7 JCSystem.commitTransaction();
```

By executing the `arrayCopyNonAtomic` within the infinite loop we know that the card will almost certainly be executing an `arrayCopyNonAtomic` operation during the card tear. You might be unlucky that the system just finished one and has not started yet on the next one. We don’t know how we could exclude this possibility completely. But by running this test scenario several times at least statistically you should get a lot of card tears during the `arrayCopyNonAtomic` operation.

Note that we have to introduce a third array `v` here. Otherwise it will not be possible to distinguish between a card tear half way during the third iteration or a card tear exactly after the first `arrayCopyNonAtomic` has completed.

3.2 Test scenarios

All test scenarios are carried out using the persistent byte arrays `t`, `u`, and `v`, initialized as follows

```java
final static short len = 0x15;
byte[] t = new byte[len];
byte[] u = new byte[len];
byte[] v = new byte[len];
Util.arrayFillNonAtomic(t,(short)0,len,(byte)0);
Util.arrayFillNonAtomic(u,(short)0,len,(byte)2);
Util.arrayFillNonAtomic(v,(short)0,len,(byte)7);
```

So when each test scenario is started, `t` contains all 0’s, `u` contains all 2’s, and `v` contains all 7’s.

After each scenario we describe the result. I.e. we describe the contents of the persistent array `t`. If we cannot explain these results we show them in italics. A dash in one of the columns means that the corresponding outcome in the other column does not occur for this card.

1 Note that you can’t wait too long to tear out the card, because most readers will have some kind of ‘time out’ system.
3.3 First conditional, then unconditional assignments

In examples 4 and 5 below we consider we consider a transaction which first modifies \( t \) using normal assignments, and then modifies \( t \) using `arrayCopyNonAtomic`.

Example 4 (ID 01, 02, 03, 04).²

```
1 // Initially \( t[i] = 0, \ u[i] = 2 \)
2 JCSystem.beginTransaction();
3 for (i = 0; i < len; i++) t[i]++; // \( t[i] \) becomes 1
4 Util.arrayCopyNonAtomic(u,(short)0,t,(short)0,len); // \( t[i] \) becomes 2
5 for (i = 0; i < len; i++) t[i]++; // \( t[i] \) becomes 3
```

Values of \( t[i] \) when we get a card tear

<table>
<thead>
<tr>
<th>ID</th>
<th>JCP</th>
<th>Palmera</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>all 0's</td>
<td>all 0's</td>
</tr>
<tr>
<td>02</td>
<td>all 0's</td>
<td>all 0's</td>
</tr>
<tr>
<td>03</td>
<td>all 0's</td>
<td>all 0's</td>
</tr>
<tr>
<td>04</td>
<td>all 0's</td>
<td>all 0's</td>
</tr>
</tbody>
</table>

Example 5 (ID 08). To consider what happens when a card tear occurs during the call of `arrayCopyNonAtomic`, we consider the following variant of Example 4, where line 4 and 5 are replaced by an infinite repetition:

```
1 // Initially \( t[i] = 0, \ u[i] = 2, \ v[i] = 7 \)
2 JCSystem.beginTransaction();
3 for (i = 0; i < len; i++) t[i]++; // \( t[i] \) becomes 1
4 while (true) {
5   Util.arrayCopyNonAtomic(u,(short)0,t,(short)0,len); // \( t[i] \) becomes 2
6   Util.arrayCopyNonAtomic(v,(short)0,t,(short)0,len); // \( t[i] \) becomes 7
7 }
```

By executing this code and producing card tears a number of times we will sometimes interrupt the calls to `arrayCopyNonAtomic`. The table below shows the contents of \( t \) we observed after several card tears:

<table>
<thead>
<tr>
<th>ID</th>
<th>JCP</th>
<th>Palmera</th>
</tr>
</thead>
<tbody>
<tr>
<td>08</td>
<td>all 0's</td>
<td>all 0's</td>
</tr>
</tbody>
</table>

So on the JCP card the array \( t \) always contained all 0's, which was what we would expect on the basis of Example ex:a, but on the Palmera card we sometimes found different contents.

We cannot explain this behaviour of the Palmera card. Of course we are not sure which of the two calls to `arrayCopyNonAtomic` we interrupt, or at which point in their execution it is interrupted, but in Example 4 the roll-back of the unfinished transaction always restores \( t \) to its state before the modification in line 3. Consequently, one would expect that in Example 5 the roll-back of an unfinished transaction always to restore \( t \) to its state before the modification in line 3.

3.4 First unconditional, then conditional assignments

In examples 6 and 7 below we consider a transaction which first modifies \( t \) using `arrayCopyNonAtomic`, and then modifies \( t \) using normal assignments:

Example 6 (ID 05, 06). Here we consider a transaction which first modifies \( t \) using `arrayCopyNonAtomic`, and then modifies \( t \) using normal assignments:

² These IDs refer to the parameter bytes being used in the applet to trigger a scenario.
// Initially t[i] = 0, u[i] = 2, v[i] = 7
JCSystem.beginTransaction();
Util.arrayCopyNonAtomic(u,(short)0,t,(short)0,len);
// t[i] becomes 2
for (i = 0; i < len; i++) t[i]++;
// t[i] becomes 3

Contents of t after we produce a card tear

<table>
<thead>
<tr>
<th>ID 05 (after line 3)</th>
<th>JCP</th>
<th>Palmera</th>
</tr>
</thead>
<tbody>
<tr>
<td>all 2’s</td>
<td></td>
<td>all 2’s</td>
</tr>
<tr>
<td>ID 06 (after line 4)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>all 2’s</td>
<td></td>
<td>all 2’s</td>
</tr>
</tbody>
</table>

So both cards restore t to the contents it had at the end of the call to arrayCopyNonAtomic.

Example 7 (ID 07). To find out what happens when a card tear occurs during the call of arrayCopyNonAtomic, we consider the following variant of Example 6, where line 3 and 4 are replaced by an infinite repetition:

```
// Initially t[i] = 0, u[i] = 2, v[i] = 7
JCSystem.beginTransaction();
while (true) {
    Util.arrayCopyNonAtomic(u,(short)0,t,(short)0,len);
    // t[i] becomes 2
    Util.arrayCopyNonAtomic(v,(short)0,t,(short)0,len);
    // t[i] becomes 7
}
```

Values of t[i] observed after card tears, in line 4 or 5:

<table>
<thead>
<tr>
<th>ID 07</th>
<th>JCP</th>
<th>Palmera</th>
</tr>
</thead>
<tbody>
<tr>
<td>some 2’s, then 7’s</td>
<td>some 2’s, then 7’s</td>
<td>some 2’s, then 7’s</td>
</tr>
<tr>
<td>some 7’s, then 2’s</td>
<td>some 7’s, then 2’s</td>
<td>some 7’s, then 2’s</td>
</tr>
<tr>
<td>only 2’s</td>
<td>only 2’s</td>
<td>only 2’s</td>
</tr>
<tr>
<td>only 7’s</td>
<td>only 7’s</td>
<td>only 7’s</td>
</tr>
<tr>
<td>some 2’s, then 0’s</td>
<td>some 2’s, then gibberish</td>
<td>some 7’s, then gibberish</td>
</tr>
<tr>
<td>some 0’s, then 7’s</td>
<td>some 7’s, then gibberish</td>
<td>only gibberish</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>some gibberish, then 2’s</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>some gibberish, then 7’s</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>some gibberish, then gibberish</td>
</tr>
</tbody>
</table>

The first five outcomes are easy to explain. The first four entries, which we observed on both cards, can be expected from producing a card tear during the invocations in line 4 or 5 (which should produce a mixture of 2’s and 7’s) or producing a card tear exactly between these invocations (which should produce all 2’s or all 7’s). To get the fifth outcome—some 2’s, then 0’s—we apparently produced a card tear when line 4 was being executed for the first time.

For the JCP cards, we can not find any explanation on how we got the sixth outcome—some 0’s, then 7’s.

For the Palmera card there is more unexplained behaviour. Here we observed that t contains some apparently gibberish data, which included values other than 0, 2, and 7, which are the only values ever assigned to the t[i].

Note that this data is not completely random. We keep observing the same sequence of unknown values each time. However we did notice that this sequence is different for different Palmera cards, and sometimes changed if we modified the code and downloaded the new applet to the card. E.g., running the code above with different length for t on one Palmera card, we observed the following values for t after card tears

\[ t = \{30, 9E, B9, AA, 94, 3D, 57, 18, 9F, 64\} \]
\[ t = \{30, 9E, B9, AA, 94, 3D, 57, 18, 9F, 64, 93, 27, 76, 49, 6 E, E7\} \]
\[ t = \{30, 9E, B9, AA, 94, 3D, 57, 18, 9F, 64, 93, 27, 76, 49, 6 E, E7, 9A, A6, CA, 01, 55\} \]

whereas on another Palmera card we observed the contents
For C code such behaviour would not really be surprising, given the vagueries of initialization and pointer arithmetic, but for Java code, let alone for single-threaded Java Card code, which should be completely deterministic, this behaviour is rather worrying!

### 3.5 First unconditional, then conditional and then some more unconditional assignments

**Example 8 (ID 09).**

```java
// Initially t[i] = 0, u[i] = 2, v[i] = 7
JCSystem.beginTransaction();
Util.arrayCopyNonAtomic(u,(short)0,t,(short)0,len); // t[i] becomes 2
for (i = 0; i<len; i++) t[i]++; // t[i] becomes 3
Util.arrayCopyNonAtomic(u,(short)0,t,(short)0,len); // t[i] becomes 2
// card tear
```

<table>
<thead>
<tr>
<th></th>
<th>JCOP</th>
<th>Palmera</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID 09</td>
<td>only 2’s</td>
<td>only 2’s</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>some 2’s, some 3’s, then 2’s</td>
</tr>
</tbody>
</table>

The only 2’s is understandable, since this is the value of \( t \) before the first conditional update. However, the other behaviour of the Palmera cards here cannot be understood. What might have been the case is that the card tear was too early and during the `arrayCopyNonAtomic` of line 5. At least that could explain a sequence of both 2’s and 3’s. However, since we have 2’s, then 3’s and then again some 2’s, this could have only caused this result if the order of copying the data need not be determined from left to right.

**Example 9 (ID 0A).**

```java
// Initially t[i] = 0, u[i] = 2, v[i] = 7
JCSystem.beginTransaction();
Util.arrayCopyNonAtomic(u,(short)0,t,(short)0,len); // t[i] becomes 2
for (i = 0; i<len; i++) t[i]++; // t[i] becomes 3
for (i = 0; i<len; i++) u[i]++; // u[i] becomes 3
Util.arrayCopyNonAtomic(u,(short)0,t,(short)0,len); // t[i] remains 3
// card tear
```

<table>
<thead>
<tr>
<th></th>
<th>JCOP</th>
<th>Palmera</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID 0A</td>
<td>only 2’s</td>
<td>only 2’s</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>some 2’s, some 3’s, then 2’s</td>
</tr>
</tbody>
</table>

As in Example 8, we cannot explain the behaviour of the Palmera card.

### 3.6 EEPROM word issues

Writing to EEPROM typically goes in terms of words, not bytes. This seems to be the reason why we see that the different blocks we get in our results are always of the same length.

**Example 10 (ID 0B).** The difference with Example 9 lies in the fact that here we do our conditional updates in a different order. Probably the two arrays \( t \) and \( u \) are not in the same words on EEPROM level and therefore the results might be different. Although it is more likely that we notice different behaviour if the card tear occurs during one of the `arrayCopyNonAtomic`'s.
// Initially t[i] = 0, u[i] = 2, v[i] = 7
JCSystem.beginTransaction();
Util.arrayCopyNonAtomic(u,(short)0,t,(short)0,len); // t[i] becomes 2
for (i = 0; i<len; i++) {
    t[i]++; // t[i] becomes 3
    u[i]++; // u[i] becomes 3
}
Util.arrayCopyNonAtomic(u,(short)0,t,(short)0,len); // t[i] remains 3
//card tear

Here the behaviour of both cards can be explained.

Example 11 (ID 0C). Here we do not modify the whole array conditionally, but only the first half of it. This test is used to check whether only the bytes really modified are backed up and restored or also the other bytes of the modified array.

The first result of only 0’s can be explained by a card tear just before the `arrayCopyNonAtomic`. The results where the first half are 0’s and the second half are filled with only 2’s or 7’s can be explained by a card tear just between the `arrayCopyNonAtomic`’s. The results where the first half are 0’s and the second half starts with 2’s and ends with 0’s can be explained by a card tear in the first execution of the `arrayCopyNonAtomic` in line 5. The results where the first half are 0’s and the second half starts with 2’s and ends with 7’s can be explained by a card tear in any execution of the `arrayCopyNonAtomic` in line 5, except the first execution. The results where the first half are 0’s and the second half starts with 7’s and ends with 2’s can be explained by a card tear in any execution of the `arrayCopyNonAtomic` in line 6.

The JCOP result with the first half 0’s and the second half starting with 0’s and ending with 7’s cannot be explained. The 7’s only appear after all 0’s in the second half have been replaced by 2’s, hence no 0’s should appear! The only explanation we can think of is that the 2’s are first replaced by 0’s before being replaced by 2’s, but we can’t believe that that is the way it is actually done.

The Palmera results containing the gibberish cannot be explained. The outcome is very similar to the one in example 5. In fact the strange bytes we see are exactly the same!
3.7 First conditional, then unconditional assignments, revisited

Examples 12 and 13 below are similar to examples 4 and 5: again we consider a transaction which first assigns to \( t \) using normal assignments, and then assigns to \( t \) using `arrayCopyNonAtomic`. The difference is that we replace \( t[i]++ \) by \( t[i]=t[i] \). Clearly this assignment has no side-effect; in fact, a compiler might optimise such assignments away. It turns out this affects the behaviour of the transaction mechanism on the JCOP cards.

**Example 12 (ID 0D, 0E).**

```
1 // Initially t[i] = 0, u[i] = 2, v[i] = 7
2 JCSystem.beginTransaction();
3 for (i = 0; i < len; i++) t[i] = t[i]; // t[i] remains 0
4 Util.arrayCopyNonAtomic(u,(short)0,t,(short)0,len); // t[i] becomes 2
5 for (i = 0; i < len; i++) t[i]++; // t[i] becomes 3
```

First we check what happens if the card tear occurs after line 4 or 5.

<table>
<thead>
<tr>
<th></th>
<th>JCOP</th>
<th>Palmera</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID 0D (after 4)</td>
<td>only 0's</td>
<td>only 0's</td>
</tr>
<tr>
<td>ID 0E (after 5)</td>
<td>only 2's</td>
<td>only 0's</td>
</tr>
</tbody>
</table>

Note that here the JCOP cards behave differently than in Example 4. Apparently the transaction mechanism on the JCOP cards notices that the assignments \( t[i]=t[i] \) have no effect, so that it decides not to ‘back-up’ the values of \( t[i] \) to roll back when a card tear occurs.

**Example 13 (ID 0F).** Now we check what happens if a card tear occurs during the call of `arrayCopyNonAtomic`. We consider the following variant of Example 12, where line 4 is replaced by an infinite repetition:

```
1 // Initially t[i] = 0, u[i] = 2, v[i] = 7
2 JCSystem.beginTransaction();
3 for (i = 0; i < len; i++) t[i] = t[i]; // t[i] remains 0
4 while (b) {
5   Util.arrayCopyNonAtomic(u,(short)0,t,(short)0,len); // t[i] become 2
6   Util.arrayCopyNonAtomic(v,(short)0,t,(short)0,len); // t[i] become 7
7 }
```

Values of \( t[i] \) after card tear in line 4 or 5:

<table>
<thead>
<tr>
<th></th>
<th>JCOP</th>
<th>Palmera</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID 0F</td>
<td>only 2's</td>
<td>only 0's</td>
</tr>
<tr>
<td></td>
<td>only 7's</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>some 7's, then 2's</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>some 2's, then 7's</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>some 2's, then 0's</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>some 0's, then 7's</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>random 7, 6, 3 and 2, then 7's, then 2's</td>
<td>-</td>
</tr>
</tbody>
</table>

Here there is some unexplainable behaviour on the JCOP card. Some 2’s and then some 0’s can be explained; apparently we produced a card tear the first time line 5 was executed. However, we cannot explain how \( t \) could contain only 0’s and 7’s. The occurrence of numbers other than 0, 2, and 7 in \( t \) observed can also not be explained.

The results of the Palmera card are understandable. The compiler doesn’t optimise and hence this test starts with conditional updates which trigger the back-up system.

\(^3\) In order to rule out the fact that this difference was caused by a different compiler or cap converter, we have installed the cap file generated by the JCOP tools also on the Palmera card. This showed the same behaviour as the cap file produced with Sun’s converter.
3.8 Some more experiments

Arrays of length 1 Due to the strange behaviour in certain test scenarios, we were interested in knowing whether this behaviour could be reproduced in case the length of the array equals one. This is an interesting case: in OwnerPIN implementations it is likely that checking a PIN code involves an `arrayCopyNonAtomic` for adjusting the -called try counter, as illustrated in Example 1.

<table>
<thead>
<tr>
<th></th>
<th>JOCOP</th>
<th>Palmera</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ex. 7 ID 07</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>D1</td>
</tr>
<tr>
<td>Ex. 5 ID 08</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Ex. 8 ID 09</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Ex. 9 ID 0A</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Ex. 11 ID 0C</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>D1</td>
</tr>
<tr>
<td>Ex. 13 ID 0F</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>

The results are similar to the results we have found using larger arrays. The JOCOP card still shows the 0 in ID 07 and ID 0C, which can only be explained by a card tear that occurred too soon. The Palmera card still shows some strange D1 in these experiments on an entry that could have only been 0, 2 or 7. Hence in particular this might indicate that these Palmera cards are not safe with respect to their implementation of OwnerPIN!

Outside transactions Although the total work for this research was inspired by our formalization of the transaction mechanism, the results triggered us to perform some checks on the `arrayCopyNonAtomic` method outside transactions. A bit to our surprise the results are again similar! The gibberish bytes for the Palmera show up again in all tests where the card tear occurs during the `arrayCopyNonAtomic`. These are the tests with ID 47, 48, 4C and 4F. The only remarkable thing here is that the original test ID 08 with the transaction didn’t show the gibberish. The JOCOP card still shows the strange combination of 0’s and 7’s in with tests ID 47 and 4F. All other results are normal.

Hence the conclusion might be that the problems are not really caused by the combination of `arrayCopyNonAtomic` and the transaction mechanism, but are inherent to the implementation of `arrayCopyNonAtomic` itself!

4 Conclusions

Below we summarise our main conclusions w.r.t. Sun’s specifications, our formalisation of the semantics of the transaction mechanism, and the strange behaviour noticed during our experimentation with actual smartcards.

Sun’s Java Card platform specification

A positive conclusion from our experiments is that, although the Java Card platform specification by Sun leaves open room for interpretation, the cards we tried actually implement the same behaviour here, and hence behave identical for examples 2 and 3 in Section 2.2. This means the ambiguity we noted in Sun’s specification can be resolved. In particular, the existing specification
If power is lost (tear) or the card is reset or some other system failure occurs while a transaction is in progress, then the JCRE shall restore to their previous values all [persistent] fields and array components conditionally updated since the previous call to `JCSystem.beginTransaction`. This makes precise what is meant by “their previous values”. This improved specification correctly predicts the outcome of examples 2 and 3 in Section 2.2. (However, it still leaves open the issue noted about the JCOP cards in the end of Example 12, where updates that have no effect, such as `t[i]=t[i]`, are apparently ignored.)

**Our formal semantics presented in [HP03]**

A negative conclusion of our experiments is that the formal description of the semantics of the transaction mechanism proposed in [HP03] is not quite right, as this assumes another interpretation of what “previous values” means. The semantics described in [HP03] assumes that

“If power is lost (tear) or the card is reset or some other system failure occurs while a transaction is in progress, then the JCRE shall restore all [persistent] fields and array components conditionally updated since the previous call to `JCSystem.beginTransaction` to the values they had directly at the moment of the previous call to `JCSystem.beginTransaction`.”

For instance, in examples 2 and 3 this will produce the wrong result. The formal semantics proposed in [HP03] could be adapted, though the technical details will become trickier.

**Strange behaviour of smartcards**

A more important negative conclusion of our experiments concerns the strange behaviour of the cards if card tears occur during calls to the non-atomic API methods `arrayCopyNonAtomic` and `arrayFillNonAtomic`. The behaviour under these circumstances is essentially completely unpredictable. Note that in the tables in Section 3 all the results written in italics are results we cannot explain.

Especially the Schlumberger cards behave very strange in these cases, with completely random data ending up in the destination array in examples 7 and 11. This strange behaviour is easy to repeat, and we believe it is a bug in the platform implementation. It is intriguing that we get up with the same sequence of apparently random data every time, but that this sequence changes if we upload a different applet to the card. This suggests that we are in fact reading out the value of some fixed page of the EEPROM here. This could be applet code, or data of the JCRE itself or some an applet. We have checked whether the sequence of apparently random data occurs in the hexadecimal representation of the unjarred cap file of our package, but this was not the case; hence we think it is unlikely that the sequence of data is part of some applet’s code. Fortunately, the random data are bytes and not references, as the API only provides the non-atomic methods for byte arrays: if the random data were references, this might break type safety.

The IBM JCOP cards clearly behave better when card tears occur during calls to `arrayCopyNonAtomic` or `arrayFillNonAtomic`. Although there are still some results we don’t understand, we cannot get completely random data in the destination array. Also, the unexplained results were hard to duplicate. In fact, in some cases it only occurred once and we never managed to repeat it.
Note that our experiments show that the implementation of the PIN code presented in Example 1 could be attacked on a Palmera card if we would be able to generate card tears at the precise moment `arrayCopyNonAtomic` is invoked: this would reset the try counter to some random value, and there is a good chance that this will allow more guesses of the PIN code than intended. However, this would require good timing of card tears, that will be hard to achieve in practice. Still, an implementation of a PIN code which does not use `arrayCopyNonAtomic` to cleverly by-pass the transaction mechanism, but which simply refuses the check of a PIN code if a transaction is in progress (the Java Card API provides methods that can check this) might be considered more secure.

At some stage during our experiments we came up with the following programming guideline to avoid the strange behaviour that we noticed:

Never use `arrayCopyNonAtomic` or `arrayFillNonAtomic` to update a persistent array during a transaction.

However, after doing also additional experiments with invoking these methods *outside* transactions (reported in Section 3.8), we realized this programming guideline did not suffice to avoid all strange behaviour, but it needed to be extended to:

Never use `arrayCopyNonAtomic` or `arrayFillNonAtomic` to update a persistent array.

This would avoid all the strange behaviour we noticed, but is of course a very strong restriction: using these methods is pointless for transient arrays and only make sense for persistent arrays, so this guideline would effectively mean that these methods should *never* be used.

### 4.1 Acknowledgements

Thanks to Marc Witteman of Riscure for his insights and Joachim van den Berg of TNO-ITSEF for repeating some of our experiments and confirming our test results.

### References


### A The applet

Below the actual code of our test applet. In the method `runTest` the different scenarios are explained. The test applet is basically a big `switch` statement. For each scenario we add a `case` based upon a unique byte that is sent from the terminal to run the test.

Performing a test requires sending of four APDUs, for example

```
00 A4 04 00 0F 6E 6F 6E 61 74 6F 6D 69 63 72 61 6E 73 41
00 01 00 00 15
00 00 00 00 01 01
00 01 00 00 15
```
The first APDU (the first line) selects the applet, the second shows the contents of the persistent byte array \( t \), the third runs one of the scenarios, and the fourth shows the contents of \( t \) again. Note that this last APDU is only used if a scenario without card tear was executed.

To automate the process we have wrote scripts for such tests, and we used the *scriptor* tool [VR03] to send these to the card. The scripts need to be modified if the length of the arrays changes, though this could be done automatically.

```java
package CopyNonAtomicTransactionTest;

import javacard.framework.*;

public class CopyNonAtomicTransactionTest extends javacard.framework.Applet {
    final static short len = 0x20;
    byte[] t = new byte[len];
    byte[] u = new byte[len];
    byte[] v = new byte[len];
    boolean b = true;

    public static final byte INS_DO_TEST = (byte) 0x00;
    public static final byte INS_GET_T = (byte) 0x01;
    public static final byte INS_RESET = (byte) 0x04;
    public static final byte INS_SELECT = (byte) 0xA4;

    public static void install(byte[] bArray, short bOffset, byte bLength) {
        (new CopyNonAtomicTransactionTest()).register(bArray, (short)(bOffset + 1), bArray[bOffset]);
    }

    public void runTest(byte ttype) {
        short i;
        try {
            if (((short) (ttype & 0x00FF) < (short)(0x40 & 0x00FF))) {
                // Tests within the transaction system
                JCSystem.beginTransaction();
            } else {
                // Tests outside the transaction system
            }

            switch (ttype) {
                case 0x01:
                    // beginTrans
                    // <card_tear/>
                    // commitTrans
                    while (b) {
                    }
                break;
                case 0x02:
                    // beginTrans
                    // t[0]++;
                    // <card_tear/>
                    // commitTrans
                    for (i = 0; i < len; i++) {
                        t[i] = (byte)(t[i] + 1);
                    }
                    while (b) {
                    }
                break;
                case 0x03:
                    // beginTrans
                    // t[0]++;
                    // arrayCopyNonAtomic(u,0,t,0,1);
```
// <card_tear/>
// commitTrans
for (i = 0; i < clen; i++) {
    t[i] = (byte)(t[i] + 1);
} Util.arrayCopyNonAtomic(u,(short)0,t,(short)0,len);
while (b) {
} break;
case 0x04:
// beginTrans
  // t[0]++;
  // arrayCopyNonAtomic(u,0,t,0,1);
  // t[0]++;
  // <card_tear/>
  // commitTrans
for (i = 0; i < clen; i++) {
    t[i] = (byte)(t[i] + 1);
} Util.arrayCopyNonAtomic(u,(short)0,t,(short)0,len);
for (i = 0; i < clen; i++) {
    t[i] = (byte)(t[i] + 1);
} while (b) {
} break;
case 0x05:
// beginTrans
  // arrayCopyNonAtomic(u,0,t,0,1);
  // <card_tear/>
  // commitTrans
Util.arrayCopyNonAtomic(u,(short)0,t,(short)0,len);
while (b) {
} break;
case 0x06:
// beginTrans
  // arrayCopyNonAtomic(u,0,t,0,1);
  // t[0]++;
  // <card_tear/>
  // commitTrans
Util.arrayCopyNonAtomic(u,(short)0,t,(short)0,len);
for (i = 0; i < clen; i++) {
    t[i] = (byte)(t[i] + 1);
} while (b) {
} break;
case 0x07:
// beginTrans
  // <card_tear>
  // arrayCopyNonAtomic(u,0,t,0,1);
  // arrayCopyNonAtomic(v,0,t,0,1);
  // <card_tear>
  // commitTrans
while (b) {
    Util.arrayCopyNonAtomic(u,(short)0,t,(short)0,len);
    Util.arrayCopyNonAtomic(v,(short)0,t,(short)0,len);
} break;
case 0x08:
// beginTrans
  // t[0]++;
  // <card_tear>
  // arrayCopyNonAtomic(u,0,t,0,1);
  // arrayCopyNonAtomic(v,0,t,0,1);
  // <card_tear>
  // commitTrans
for (i = 0; i < clen; i++) {
    t[i] = (byte)(t[i] + 1);
} while (b) {
    Util.arrayCopyNonAtomic(u,(short)0,t,(short)0,len);
    Util.arrayCopyNonAtomic(v,(short)0,t,(short)0,len);
} break;
case 0x09:
// beginTrans
  // arrayCopyNonAtomic(u,0,t,0,1);
  // t[0]++;
  // <card_tear>
17

// arrayCopyNonAtomic(u,0,t,0,1);
// <card_tear/>
commitTrans
Util.arrayCopyNonAtomic(u,(short)0,t,(short)0,len);
for (i = 0; i<len; i++) {
    t[i] = (byte)(t[i] + 1);
}
Util.arrayCopyNonAtomic(u,(short)0,t,(short)0,len);
while (b) {
    break;
}
case 0x0a:
    // beginTrans
    // arrayCopyNonAtomic(u,0,t,0,1);
    // t[0]++; (only half array)
    // <card_tear/>
    // commitTrans
    Util.arrayCopyNonAtomic(u,(short)0,t,(short)0,len);
    for (i = 0; i<len; i++) {
        t[i] = (byte)(t[i] + 1);
    }
    for (i = 0; i<len; i++) {
        u[i] = (byte)(u[i] + 1);
    }
    Util.arrayCopyNonAtomic(u,(short)0,t,(short)0,len);
    while (b) {
        break;
    }

break;
// t[0]++;
// <card_tear>
// commitTrans
for (i = 0; i < len; i++) {
    t[i] = t[i];
}
Util.arrayCopyNonAtomic(u, (short)0, t, (short)0, len);
for (i = 0; i < len; i++) {
    t[i] = (byte)(t[i] + 1);
}
while (b) {
    break;
}

case 0x0f:
// beginTrans
// t[0] = t[0];
// <card_tear>
// arrayCopyNonAtomic(u,0,t,0,1);
// arrayCopyNonAtomic(v,0,t,0,1);
// <card_tear>
// commitTrans
for (i = 0; i < len; i++) {
    t[i] = t[i];
}
while (b) {
    Util.arrayCopyNonAtomic(u, (short)0, t, (short)0, len);
    Util.arrayCopyNonAtomic(v, (short)0, t, (short)0, len);
}
    break;

case 0x41:
// <card_tear/>
while (b) {
    break;
}

case 0x42:
// t[0]++;
// <card_tear/>
for (i = 0; i < len; i++) {
    t[i] = (byte)(t[i] + 1);
}
while (b) {
    break;
}

case 0x43:
// t[0]++;
// arrayCopyNonAtomic(u,0,t,0,1);
// <card_tear>
for (i = 0; i < len; i++) {
    t[i] = (byte)(t[i] + 1);
}
Util.arrayCopyNonAtomic(u, (short)0, t, (short)0, len);
for (i = 0; i < len; i++) {
    t[i] = (byte)(t[i] + 1);
}
while (b) {
    break;
}

case 0x44:
// t[0]++;
// arrayCopyNonAtomic(u,0,t,0,1);
// t[0]++;
// <card_tear/>
for (i = 0; i < len; i++) {
    t[i] = (byte)(t[i] + 1);
}
Util.arrayCopyNonAtomic(u, (short)0, t, (short)0, len);
for (i = 0; i < len; i++) {
    t[i] = (byte)(t[i] + 1);
}
while (b) {
    break;
}

case 0x45:
// arrayCopyNonAtomic(u,0,t,0,1);
// <card_tear/>
Util.arrayCopyNonAtomic(u, (short)0, t, (short)0, len);
while (b) {
    break;
}

case 0x46:
// arrayCopyNonAtomic(u,0,t,0,1);
// t[0]++;
// <card_tear/>
Util.arrayCopyNonAtomic(u,(short)0,t,(short)0,len);
for (i = 0; i<len; i++) {
    t[i] = (byte)(t[i] + 1);
}
while (b) {
    break;
}
case 0x47:
    // beginTrans
    // \langle card\tear\rangle
    // arrayCopyNonAtomic(u,0,t,0,1);
    // arrayCopyNonAtomic(v,0,t,0,1);
    // \langle/ card\tear\rangle
    // commitTrans
    while (b) {
        Util.arrayCopyNonAtomic(u,(short)0,t,(short)0,len);
        Util.arrayCopyNonAtomic(v,(short)0,t,(short)0,len);
    }
    break;

case 0x48:
    // beginTrans
    // arrayCopyNonAtomic(u,0,t,0,1);
    // t[0]++;
    // arrayCopyNonAtomic(u,0,t,0,1);
    // \langle card\tear\rangle
    for (i = 0; i<len; i++) {
        t[i] = (byte)(t[i] + 1);
    }
    while (b) {
        Util.arrayCopyNonAtomic(u,(short)0,t,(short)0,len);
        Util.arrayCopyNonAtomic(v,(short)0,t,(short)0,len);
    }
    break;

case 0x49:
    // beginTrans
    // arrayCopyNonAtomic(u,0,t,0,1);
    // t[0]++;
    // arrayCopyNonAtomic(u,0,t,0,1);
    // \langle card\tear\rangle
    for (i = 0; i<len; i++) {
        t[i] = (byte)(t[i] + 1);
    }
    for (i = 0; i<len; i++) {
        u[i] = (byte)(u[i] + 1);
    }
    Util.arrayCopyNonAtomic(u,(short)0,t,(short)0,len);
    while (b) {
    }
    break;

case 0x4a:
    // arrayCopyNonAtomic(u,0,t,0,1);
    // t[0]++;
    // u[0]++;
    // arrayCopyNonAtomic(u,0,t,0,1);
    // \langle card\tear\rangle
    Util.arrayCopyNonAtomic(u,(short)0,t,(short)0,len);
    for (i = 0; i<len; i++) {
        t[i] = (byte)(t[i] + 1);
    }
    for (i = 0; i<len; i++) {
        u[i] = (byte)(u[i] + 1);
    }
    Util.arrayCopyNonAtomic(u,(short)0,t,(short)0,len);
    while (b) {
    }
    break;

case 0x4b:
    // arrayCopyNonAtomic(u,0,t,0,1);
    // t[0]++;
    // u[0]++;
    // arrayCopyNonAtomic(u,0,t,0,1);
    // \langle card\tear\rangle
    Util.arrayCopyNonAtomic(u,(short)0,t,(short)0,len);
    for (i = 0; i<len; i++) {
        t[i] = (byte)(t[i] + 1);
    }
    for (i = 0; i<len; i++) {
        u[i] = (byte)(u[i] + 1);
    }
    Util.arrayCopyNonAtomic(u,(short)0,t,(short)0,len);
    while (b) {
    }
}
break;
case 0x4c:
  // (0)[++] (only half array)
  // <card_tear>
  // arrayCopyNonAtomic(u,0,t,0,1);
  // arrayCopyNonAtomic(v,0,t,0,1);
  // <card_tear>
  for (i = 0; i < (short)(len/2); i++) {
    t[i] = (byte)(t[i] + 1);
  }
  while (b) {
    Util.arrayCopyNonAtomic(u,(short)0,t,(short)0,len);
    Util.arrayCopyNonAtomic(v,(short)0,t,(short)0,len);
  }
break;
case 0x4d:
  // beginTrans
  // t[0] = t[0];
  // <card_tear>
  // commitTrans
  for (i = 0; i < len; i++) {
    t[i] = t[1];
  }
  Util.arrayCopyNonAtomic(u,(short)0,t,(short)0,len);
  while (b) {
  }
break;
case 0x4e:
  // beginTrans
  // t[0] = t[0];
  // <card_tear>
  // commitTrans
  for (i = 0; i < len; i++) {
    t[i] = t[i];
  }
  Util.arrayCopyNonAtomic(u,(short)0,t,(short)0,len);
  for (i = 0; i < len; i++) {
    t[i] = (byte)(t[i] + 1);
  }
  while (b) {
  }
break;
case 0x4f:
  // t[0] = t[0];
  // <card_tear>
  // arrayCopyNonAtomic(u,0,t,0,1);
  // arrayCopyNonAtomic(v,0,t,0,1);
  // <card_tear>
  for (i = 0; i < len; i++) {
    t[i] = t[1];
  }
  while (b) {
    Util.arrayCopyNonAtomic(u,(short)0,t,(short)0,len);
    Util.arrayCopyNonAtomic(v,(short)0,t,(short)0,len);
  }
break;
default:
  // No card tears
  // beginTrans
  // t[0]++;
  // <card_tear>
  // commitTrans
  for (i = 0; i < len; i++) {
    t[i] = (byte)(t[i] + 1);
  }
  Util.arrayCopyNonAtomic(u,(short)0,t,(short)0,len);
  for (i = 0; i < len; i++) {
    t[i] = (byte)(t[i] + 1);
  }
  }
if ((short)((ttype & 0x00FF) < (short)(0x40 & 0x00FF)) {
  // Tests within the transaction system
  JCSystem.commitTransaction();
} else {
// Tests outside the transaction system

} catch (ArithmeticException e) {
    ISOException.throwIt((short)0x7000);
} catch (ArrayStoreException e) {
    ISOException.throwIt((short)0x7100);
} catch (ClassCastException e) {
    ISOException.throwIt((short)0x7200);
} catch (ArrayIndexOutOfBoundsException e) {
    ISOException.throwIt((short)0x7300);
} catch (IndexOutOfBoundsException e) {
    ISOException.throwIt((short)0x7400);
} catch (NegativeArraySizeException e) {
    ISOException.throwIt((short)0x7500);
} catch (NullPointerException e) {
    ISOException.throwIt((short)0x7600);
} catch (SecurityException e) {
    ISOException.throwIt((short)0x7700);
} catch (APDUException e) {
    ISOException.throwIt((short)(0x7800 | e.getReason()));
} catch (ISOException e) {
    ISOException.throwIt((short)(0x7900 | e.getReason()));
} catch (PINException e) {
    ISOException.throwIt((short)(0x7a00 | e.getReason()));
} catch (SystemException e) {
    ISOException.throwIt((short)(0x7b00 | e.getReason()));
} catch (TransactionException e) {
    ISOException.throwIt((short)(0x7c00 | e.getReason()));
} catch (CardRuntimeException e) {
    ISOException.throwIt((short)(0x7d00 | e.getReason()));
} catch (RuntimeException e) {
    ISOException.throwIt((short)(0x7e00 | e.getReason()));
} catch (Exception e) {
    ISOException.throwIt((short)(0x7f00 | e.getReason()));
}

public void reset() {
    Util.arrayFillNonAtomic(t, (short)0,len,(byte) 0);
    Util.arrayFillNonAtomic(u, (short)0,len,(byte) 2);
    Util.arrayFillNonAtomic(v, (short)0,len,(byte) 7);
}

public void process(APDU apdu) {
    byte[] buf = apdu.getBuffer();
    switch(buf[ISO7816.OFFSET_INS]){
    case (byte)0xb1:
        break;
    case (byte)0x05:
        break;
    case (byte)0x06:
        break;
    case (byte)0x07:
        break;
    case (byte)0x08:
        break;
    case (byte)0x09:
        break;
    case (byte)0x0a:
        break;
    case (byte)0x0b:
        break;
    case (byte)0x0c:
        break;
    case (byte)0x0d:
        break;
    case (byte)0x0e:
        break;
    case (byte)0x0f:
        break;
    case (byte)0x10:
        break;
    case (byte)0x11:
        break;
    case (byte)0x12:
        break;
    case (byte)0x13:
        break;
    case (byte)0x14:
        break;
    case (byte)0x15:
        break;
    case (byte)0x16:
        break;
    case (byte)0x17:
        break;
    case (byte)0x18:
        break;
    case (byte)0x19:
        break;
    case (byte)0x1a:
        break;
    case (byte)0x1b:
        break;
    case (byte)0x1c:
        break;
    case (byte)0x1d:
        break;
    case (byte)0x1e:
        break;
    case (byte)0x1f:
        break;
    case (byte)0x20:
        break;
    case (byte)0x21:
        break;
    case (byte)0x22:
        break;
    case (byte)0x23:
        break;
    case (byte)0x24:
        break;
    case (byte)0x25:
        break;
    case (byte)0x26:
        break;
    case (byte)0x27:
        break;
    case (byte)0x28:
        break;
    case (byte)0x29:
        break;
    case (byte)0x2a:
        break;
    case (byte)0x2b:
        break;
    case (byte)0x2c:
        break;
    case (byte)0x2d:
        break;
    case (byte)0x2e:
        break;
    case (byte)0x2f:
        break;
    case (byte)0x30:
        break;
    case (byte)0x31:
        break;
    case (byte)0x32:
        break;
    case (byte)0x33:
        break;
    case (byte)0x34:
        break;
    case (byte)0x35:
        break;
    case (byte)0x36:
        break;
    case (byte)0x37:
        break;
    case (byte)0x38:
        break;
    case (byte)0x39:
        break;
    case (byte)0x3a:
        break;
    case (byte)0x3b:
        break;
    case (byte)0x3c:
        break;
    case (byte)0x3d:
        break;
    case (byte)0x3e:
        break;
    case (byte)0x3f:
        break;
    default:
        ISOException.throwIt(ISO7816.SW_INS_NOT_SUPPORTED);
    }
}