Tearing Java Cards

Engelbert Hubbers, Wojciech Mostowski, and Erik Poll

Security of Systems (SoS) group
Department of Computing Science
Radboud University Nijmegen
The Netherlands
{E.Hubbers,W.Mostowski,E.Poll}@cs.ru.nl

Abstract. This paper reports on investigations into the Java Card *transaction mechanism*, especially on the interaction with so-called *non-atomic methods* in the Java Card API. This work started with efforts to develop a formalisation of the transaction mechanism that could be used to formally verify the correctness of applications that use these mechanisms to protect themselves from card tears—the sudden loss of power caused by removing a smart card from a terminal. During work to formalise the Java Card platform we came across ambiguities in the official specification, and subsequent experiments with real cards revealed that behaviour of cards varies a lot, and some Java Cards fail to meet the official specification. We will discuss the outcome of our experiments with real cards and attempts to formalise the official specifications. In particular, we show how we can break the security of the reference implementation of PIN objects on some smart cards, and how our formal specification can be used to verify the behaviour of Java Card code, even in the presence of card tears, using the KeY program verifier or using model checking.

1 Overview

One of the more complicated features of the Java Card platform is the *transaction mechanism*, which is provided to protect applications from the effect of card tears—the sudden loss of power due to removing the card from a terminal, which may occur at any moment. The transaction mechanism allows the programmer to enforce the atomicity of arbitrary Java Card code blocks, which in turn guarantees data consistency on the card. On top of that, two *non-atomic* API methods are provided (`arrayCopyNonAtomic` and `arrayFillNonAtomic`) that allow the programmer to by-pass the transaction mechanism. A possibility to by-pass the transaction mechanism can be important from the security point of view, as will be explained later using the standard example of a PIN try counter.

During our research in formal program verification [13, 1] we made an attempt to formally specify the behaviour of the Java Card transaction mechanism [3,

18, 11]. A careful study of different versions (2.1.1 and 2.2.2) of the official Sun[1] JAVA CARD specification [20, 22] lead us to the following observations:

- some very specific transaction use scenarios involving non-atomic methods are not covered in the specification,
- recent versions of the JAVA CARD specification introduce deliberate under-specification—non-deterministic behaviour of the transaction mechanism in combination with non-atomic methods and card tears is *officially* allowed.

Thus we performed an experimental study on a few cards (7 cards from 4 different manufactures) to see how the reality relates to the official specification. The result of the study clarified the missing parts of the specification, but, more importantly, also showed that the behaviour of the cards varies a lot when non-atomic methods are considered, and that some cards fail to meet the official specification, despite the freedom introduced by the under-specification.

For us, the results of the experimental study have two major consequences:

- clarified semantics of the transaction mechanism allowed us to fully formalise transactions and non-atomic methods in the KeY system—a formal program verifier for JAVA and JAVA CARD programs. This allows us to formally verify behavioural properties of programs that involve the transaction mechanism and non-atomic methods,
- the non-determinism allowed by the official JAVA CARD specification and non-compliant behaviour of some of the cards opens possibilities for cheap fault injection attacks. In particular, it is possible to break the (naïve version of) reference implementation of the `OwnerPIN` class, as we will demonstrate. To prevent such attacks, more defensive programming techniques are needed. To prove the correctness of such defensive programs with respect to card tears, we use the well-established formal verification technique of model checking.

The rest of this papers is organised as follows. Section 2 gives an overview of the JAVA CARD transaction mechanism and quotes the official JAVA CARD specification on the most important issues. Section 3 elaborates on the questions left open by the official specification. Section 4 describes our experimental study and its results. Section 5 discusses the PIN try counter example and the verification of behavioural properties of the `OwnerPIN` reference implementation. Section 6 elaborates on the possible fault injection attacks, countermeasures in the form of defensive programming against such attacks, and shows how model checking tools can be used to check the adequacy of such countermeasures. Finally, Section 7 summarises the paper.

2 JAVA CARD Transaction Mechanism

The memory model of JAVA CARD [7, 22] differs slightly from JAVA's model. In smart cards there are two kinds of writable memory: persistent (storage) memory (EEPROM), which holds its contents between card sessions, and transient

[1] `http://java.sun.com`

(scratch-pad) memory (RAM), whose contents disappear when power loss occurs, in particular when the card is removed from the card reader (card *tear*). Thus every memory element in Java Card (variable or object field) is either persistent or transient. By default, all objects are created in persistent memory. Thus, in Java Card all assignments like `o.attr = 2`, `this.a = 3`, and `arr[i] = 4` have permanent character, i.e. the assigned values will be kept after the card loses power. Additionally, an array (its contents) can also be a allocated in transient memory by calling a certain method from the Java Card API (e.g. `makeTransientByteArray`) to create the array. Finally, all local variables and some of the system owned arrays (such as the APDU buffer) are transient.

Java Card provides a transaction mechanism to perform atomic updates to persistent memory even in the case of a card tear, so that the consistency of the persistent data can be preserved. The Java Card API provides three native methods to achieve this, namely `beginTransaction`, `commitTransaction`, and `abortTransaction` in the class `JCSystem`. Any updates to persistent memory between `beginTransaction` and `commitTransaction` are guaranteed to be atomic. So after invoking `beginTransaction` any updates to persistent memory are conditional, in the sense that if there is a card tear before the subsequent invocation of `commitTransaction`, these updates will be rolled back. Any updates to persistent memory are also rolled back if after a call to `beginTransaction` the method `abortTransaction` is called explicitly. To quote Sun's JCRE specification, version 2.2.2 [22, 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 `beginTransaction`."

The situation is complicated by the fact that there are two *non-atomic* methods in the Java Card API which by-pass the transaction mechanism, namely `arrayCopyNonAtomic` and `arrayFillNonAtomic` in the class `Util`. To quote Sun's Java Card API specification [21] for these methods:

> "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 *[fill]* operation."

Not surprisingly, both `arrayCopyNonAtomic` and `arrayFillNonAtomic` are native; it would be impossible to implement this behaviour directly in the Java Card language.

There are two reasons why one might want to use the non-atomic methods: efficiency and security. Regarding efficiency, if an array can be left in a partially modified state (in particular, all transient arrays), updating it without involving the transaction mechanism will result in a faster operation of the applet. Regarding security, because the non-atomic methods by-pass the

transaction mechanism, they should be used for operations that—for security reasons—should never be rolled back. In some applications, certain data has to be updated unconditionally even when a transaction is in progress. The typical example concerns PIN try counters. Such a counter, associated with a PIN code, is decremented each time the user enters an incorrect guess of the PIN code, so that the card can "shut down" if too many incorrect guesses are done. By calling the PIN verification routine inside a transaction and deliberately aborting that transaction, the update to the try counter would be rolled back. This would be a major security breach, as it gives an attacker an infinite number of tries to guess the PIN code—the try counter would never be decremented. To avoid this situation a non-atomic method should be used to exclude any changes to try counter from any transaction that may be in progress..

In fact, one of the early reference implementations (included in the Sun Java Card Development Kit 2.0) of the `OwnerPIN` class did not take the possibility of a transaction into account: the counter was decreased in a conditional way and subjected to a transaction roll-back:[2]

```
public class OwnerPIN implements PIN {
  byte triesLeft;

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

This was corrected in the Java Card 2.1.1 reference implementation as follows:[3]

```
public class OwnerPIN implements PIN {
  byte[] triesLeft = new byte[1];
  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);
    ...
  }
}
```

[2] In most examples in this paper we skip some details to improve readability. In particular, we omit the `short`/`byte` casts required in Java Card programs.

[3] Again, this is not an *exact* quote from the reference implementation (for example, here some methods are in-lined), but the code is equivalent.

Moreover, newer versions (2.2 onwards) of the Java Card specification, introduced a disclaimer stating that the combination of the transaction mechanism and non-atomic methods may give unpredictable results in the case the transaction is aborted:

> "**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. *[caused by a card tear or a call to* ***`abortTransaction`****]*"

Note, that this statement by itself already has far-reaching consequences. It means that using a non-atomic method to by-pass the transaction mechanism is in general not safe! In particular, a card tear during the `arrayCopyNonAtomic` call in the example implementation of the `check` method in `OwnerPIN` above could leave the try counter with a random, possibly large, value. We will concentrate on this issue later in Section 6 and show how such problems can be avoided by defensive programming.

3 Open Questions in the Java Card Specification

In the course of trying to give a formal description of the Java Card transaction mechanism [3, 11, 18], we came across one issue that is not clear in the official Java Card specification. Namely, it does not state what should happen if a persistent array is updated with a regular assignment (conditionally) *and* with a non-atomic method (unconditionally) within the *same* transaction, and that transaction is aborted. For example, what is the value of `a[0]` at the end of execution of the two program blocks presented in Figure 1? Admittedly, such examples are very contrived; using both normal updates and the non-atomic methods to update the same array inside a transaction is not something one would expect to happen in normal Java Card code. Still, the language specification should be unambiguous for any legal Java Card program. After all, there are no guarantees that malicious or simply badly-written applets will not contain strange coding patterns.

```
a[0] = 0;
JCSystem.beginTransaction();
  // conditional update:
  a[0] = 1;
  // unconditional update a[0] = 2:
  Util.arrayFillNonAtomic(a,0,1,2);
JCSystem.abortTransaction();
// a[0] == ?
```

```
a[0] = 0;
JCSystem.beginTransaction();
  // unconditional update a[0] = 2:
  Util.arrayFillNonAtomic(a,0,1,2);
  // conditional update:
  a[0] = 1;
JCSystem.abortTransaction();
// a[0] == ?
```

Fig. 1. Mixing conditional and unconditional updates within one transaction.

Some other questions that we would like to be able to answer are the following. Does the unpredictability implied by non-atomic method calls inside a transaction indeed take place on real cards? If so, does it happen for all the cards, or only some? How unpredictable/random array elements can get? Possibly, do the cards reveal some other problems associated with non-atomic methods? If so, what are the consequences? Finally, we also find the phrase "contents of an array component" unclear. Does this mean that only one array element (component) associated with *one* array index can be left in an undefined state, or can the whole array be randomised?

4 Experiments

Given the questions about the transaction mechanism and non-atomic methods raised in the previous section, we carried out experiments with test applet executing on smart cards from different manufacturers. The experiments are described at length in [12]. During the course of further research we tested some more cards—currently 7 cards from 4 different manufacturers have been tested.

The test applet simply runs all sorts of combinations of the transaction mechanism and non-atomic methods: first conditional, then unconditional updates inside a transaction, vice-versa, only unconditional updates inside a transaction followed by a card tear, etc. The test array that is modified with a non-atomic method is always persistent. To generate a transaction abort at specific program points we either used the method `abortTransaction` or we inserted a non-terminating loop (`while(true){}`) at a certain program point, and did a physical card tear.

Part of the test was to find out what actually happens to array contents if a card tear occurs *during* an invocation of the non-atomic methods. Clearly, timing card tears to do this is more complicated, as the trick used above no longer works. One possibility would be to use special hardware, which interrupts the power supply at a precise moment in time, e.g. after a given number of CPU cycles or when detecting a certain behaviour on a side channel. Instead, we used the simpler trick of having applets that execute an infinite loop containing only calls to non-atomic methods, so that a physical card tear is very likely to occur during the execution of such a method. By repeating the experiments and collecting the results we were sure to include observations of interruptions during the invocation of the non-atomic methods.

4.1 Test Results

Over one hundred test combinations were run on each card giving large amounts of test data. Describing all the fine details of these results is beyond the scope of this paper, however the interesting results can be divided into four main categories:

(a) a transaction abort, either by an invocation of `abortTransaction` or by a physical card tear, *after* conditional and unconditional (non-atomic) updates inside a transaction;

(b) a transaction abort, either by an invocation of `abortTransaction` or by a physical card tear, *after* a call of a non-atomic method inside a transaction;
(c) a transaction abort by a physical card tear *during* a call of a non-atomic method *inside* a transaction;
(d) finally, a transaction abort by a physical card tear *during* a call of a non-atomic method *outside* of a transaction.

We discuss these categories in more detail below.

(a) Abort After Conditional & Unconditional Update In Transaction. The result of this test was supposed to tell us what happens to the value of `a[0]` in programs from Figure 1. It turns out that the value of `a[0]` is rolled back to 0 for the program on the left, and for the program on the right the value of `a[0]` is rolled back to 2. This behaviour is consistent for all the tested cards. The explanation for this behaviour would be the following. 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. Our experiments suggest that back-up copies of old values of data are made directly prior to the first conditional update in the transaction.

This would suggest that the official JAVA CARD specification could be refined to eliminate the ambiguity as follows:

> "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 elements conditionally updated since the previous call to `beginTransaction` to *the values they had directly prior to their first conditional update after the previous call to* `beginTransaction`.

Admittedly, this contradicts the other part of the specification stating that any unconditional (non-atomic) update inside a transaction followed by an abort can give unpredictable results, in which case we cannot really talk about a transaction roll-back—the contents of memory in question would neither be the old value (roll-back), nor the new value (no roll-back), it should simply be considered random. But let us first see what were the other test results.

Finally, the result of this test also means that a non-atomic method exhibits its non-atomic feature only if the array it modifies have not yet been conditionally updated within the same transaction. That is, non-atomic methods do not override any conditional updates already performed inside a transaction.

(b) Abort After Non-Atomic Method Inside Transaction. This test answers the question what happens if a transaction is started, then the persistent data is updated with a non-atomic method, and then, after the completion of the non-atomic call, any kind of transaction abort happens. It turns out that

the memory contents that were modified with the non-atomic method call is preserved, that is, the changes caused by the non-atomic call are kept. This result is also consistent for all the cards. This would suggest that, as far as *this* test is considered, the "unpredictable" results allowed by the official JAVA CARD specification do not take place—completed calls to non-atomic methods, even followed by an abort, do not cause unpredictable results.

(c) Abort During Non-Atomic Method Inside Transaction. The previous test allowed all the non-atomic method calls to complete before the transaction was aborted. So what happens when the abort happens *in the middle* of a non-atomic method call? Here the results differ from card to card. Assume that a contents of an array before the test is all 2's, and that the non-atomic method call tries to update all the elements to 7 when the card is teared. The following three categories of the test results can be given:

- The test array contains some 2's and some 7's, or only 2's, or only 7's. No unpredictable array contents here. Only one card gave such a result.
- Like the first one, but the array can also contain 0's. This 0 value here represents the "unpredictable" array contents. Three cards gave a result like this.
- Like the first one, but the array can also end up with "random" values. The values are random only seemingly, because the repetition of the test gives always the same byte sequence, for example, for one of the cards the test array contained the following data:

  ```
  DB 8C 07 89 AC 02 F8 07 C1 02 46 4D 47 E0 88 02 D3 DD C7 9B
  ```

  The remaining three out of the seven cards exhibited such behaviour.

This last result exemplifies the unpredictable behaviour mentioned in the official specification in "full flavour". What is worrying is the pseudo-randomness of the data, which would suggest that the test array actually contains a memory footprint of the card.

Furthermore, after this test it is still not clear what "contents of an array component" means. If the specification actually allows only for one array element to have an unpredictable value, then this result suggests non-compliance of some of the cards to the standard. But, as said, this issue is subject to the interpretation of the mentioned phrase and it as well may simply mean "whole contents of an array".

This behaviour suggests that card tears may be a way to attack JAVA CARD code that uses the non-atomic methods, as card tears can then pollute the persistent memory, effectively providing a fault injection. Section 6 discusses such experiments on implementations of the `OwnerPIN` class.

(d) Abort Tear During Non-Atomic Method Outside Transaction. Finally, we also wanted to test what happens if a card tear occurs during a non-atomic method call on a persistent array when there is *no* transaction in progress.

To our surprise, the results, for each of the cards, are exactly the same as corresponding results in the previous test. It means that the implementation of non-atomic methods does not differentiate between transaction and non-transaction context. More importantly, this behaviour should be considered to be a *bug*. The official specification does not mention anywhere the possibility of an unpredictable array contents when there is no transaction in progress and a tear occurs during a non-atomic method call. It does allow an array to be left in a *partially modified* state, but we do not believe that an array full of random data (card memory footprint) qualifies as "partially modified". Neither we can agree that an array filled with 0's should be considered as partially modified. As explained in the previous result, 0 is neither the previous value of an array element, nor the new value. Depending on the application, a persistent array that is left with 0 values may lead to a security breaching state.

This suggest that the disclaimer added in the specification of Java Card version 2.2 that we mentioned on page 2 should be generalised further by removing the restriction "while a transaction is in progress", i.e.

> "**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. *[caused by a card tear or a call to* `abortTransaction`*]*"

Non-Atomic Can Be Atomic. The tests just described revealed an interesting fact: on some cards, non-atomic methods are in fact atomic! They do by-pass the transaction mechanism, so that the updates they make are not subject to transaction roll-backs. However the array update they perform unconditionally is in itself atomic—either all elements are updated or none. This suggests the term "non-atomic" may be a bit misleading, and calling these methods "unrollbackable" would be much better. A correct but somewhat impractical name for `arrayCopyNonAtomic` would be `arrayCopyPossiblyNonAtomicDefinitelyUnRollbackable`.

5 Formal Model and Verification of Behavioural Properties

The results of the first test allowed us to clarify the ambiguity in the official Java Card specification regarding the use of conditional and unconditional updates within one transaction. This clarification was necessary to finalise the behavioural formalisation of the transaction mechanism in the KeY system. The KeY system [1] is a highly automated interactive program verifier for Java and Java Card programs. By employing mathematical rigour and different kinds of logics formal verification techniques can provide very high level of assurance with respect to absence of bugs. In case of the KeY system, the logical basis is a special version of Dynamic Logic [2]. The specification front-end of KeY is either

Java Modeling Language (JML) [14], Object Constraint Language (OCL) [23], or simply Dynamic Logic.

The full formalisation of the JAVA CARD transaction mechanism [3, 18] allows us to prove behavioural properties of JAVA CARD programs in the context of transactions [10]. Take, for example the reference implementation of the `check` method in the `OwnerPIN` class:

```
public boolean check(byte[] pin, short offset, byte length) {
   setValidatedFlag(false);
   if(getTriesRemaining() == 0)
      return false;
   temps[0] = triesLeft[0] - 1;
   Util.arrayCopyNonAtomic(temps, 0, triesLeft, 0, 1);
   if(length != pinSize)
      return false;
   if(Util.arrayCompare(this.pin, 0, pin, offset, length) == 0) {
      setValidatedFlag(true);
      triesLeft[0] = tryLimit;
      return true;
   }
   return false;
}
```

What we would like to prove is that this implementation indeed satisfies the strong security requirement from the JAVA CARD API documentation, namely, that the value of the try counter is decreased despite any transaction that may be in progress. Constructing an appropriate Dynamic Logic formula expressing this property is fairly simple, and the KeY system can prove it automatically in less than one minute time. A similar attempt to prove this result for the old reference implementation fails, as it should, because it does not by-pass the transaction mechanism, as discussed in Section 2.

Our formal model is only correct under the assumption that a direct call to `abortTransaction` will not leave any arrays in an unpredictable state. Even though this is not guaranteed by the JAVA CARD specification, the test results described in the previous section suggest that this indeed is the case, at least for the seven cards that we tested.

When an abort by a card tear is considered, our formal model can also be used, but under much stronger assumption, namely, that a card tear that occurs during a non-atomic method call can leave the array in a *strictly* partially modified state (either old values or new values in the array, *no* zero or random values), but not in an unpredictable state. We consider this assumption *strong*, because in reality only one card exhibited such behaviour. Under this assumption we can also formally reason about card tears with the KeY system, that is, prove properties about the program behaviour in cases when an unexpected transaction abort caused by a card tear occurs.

However, if we assume the unpredictable behaviour exhibited by some of the cards, the KeY system cannot formally reason about card tears. Moreover,

the quoted implementation of the `check` method is highly insecure. To make it secure, defensive programming techniques are needed, and a different formal verification technique (model checking) can be used to verify the robustness of the implementation in the context of intermittent card tears and unpredictability of the arrays modified by non-atomic methods. The next section is devoted to this subject.

6 Fault Injection Attacks by Card Tearing: Prevention and Verification

The unpredictability and faulty behaviour of some of the cards open possibilities for easy fault injection attacks. A properly timed card tear during a non-atomic method call can leave random values in card's persistent memory. For example, if a card tear occurs during the call to `arrayCopyNonAtomic` in the implementation of the `check` method, the `OwnerPIN` object may be left with a large value of the try counter. To put this theory to test, we wrote a simple test applet that included the reference implementation of the `OwnerPIN` class given earlier. By performing physical card tears during invocations of the `check` method on PIN objects of this class, it did not take long before we ended up with PIN objects containing a try counter value over 100, meaning we could try the `check` method on that object over 100 times.[4]

That convinced us that the faulty implementation of the non-atomic methods is indeed an issue. But what about the (possibly native) implementation of the `OwnerPIN` class provided as part of the card's built-in implementation of the Java Card API, in card's ROM? It definitely has to use some mechanism similar to the non-atomic methods to by-pass the transaction mechanism. Can it also be exploited in a similar way? We ran the same test applet, but this time we used the built-in implementation of the `OwnerPIN` class. So far, we did not manage to succeed in breaking the built-in implementation[5]. That can mean two things:

- most likely, the built-in implementation is indeed secure—either has its own (native), safe way of by-passing the transaction mechanism, or it does use a non-atomic method in a defensive way;
- less likely, we simply did not (yet) succeed in our attack attempts. After all, the card tear has to be properly timed, and our card tears are simply timed randomly.

While exploring different implementations of the `OwnerPIN` class one more issue came up. The Java Card API specification of `OwnerPIN` requires *all* operations performed by the `check` method on the state of the `OwnerPIN` object to be excluded from any transaction that may be in progress. However, the reference

[4] Of course, any properly defensive implementation of PIN object would detect such illegal values of the try counter and block further operations on the card!

[5] We would *not* be publishing these results if we had been able break any built-in implementation of `OwnerPIN` classes!

implementation of the `check` method given earlier (note again, this is an equivalent of Sun's official reference implementation for JAVA CARD 2.1.1) updates the try counter *conditionally* (`triesLeft[0] = tryLimit;`). To see whether the cards adhere to the specification in this respect we wrote yet another simple test:

```
pin.resetAndUnblock();
byte triesBefore = pin.getTriesRemaining();
JCSystem.beginTransaction();
  pin.check(correct_pin, ...);
JCSystem.abortTransaction();
if(pin.getTriesRemaining() == triesBefore - 1) {
  // report: maxing is conditional
}else{
  // report: maxing is unconditional
}
```

It turns out that most of the cards do the 'maxing' of the try counter when a correct PIN is entered unconditionally, following the specification. Only one card maxed the counter conditionally. Thus, we discovered another inconsistency of a card implementation with respect to the JAVA CARD specification.

Conditionally changing the try counter when the correct PIN is entered may seem a smaller security risk than conditionally changing the try counter when an incorrect PIN is entered, or not seem a security risk at all. After all, only the latter might be exploited by an attacker to get additional guesses of the PIN code. However, the former could lead to Denial-of-Service attack where the card blocks even though the correct PIN is entered.

6.1 Preventing "Non-Atomic" Fault Injections

The natural next step after breaking the reference implementation of `OwnerPIN` is to try to give a non-native implementation of a secure try counter despite the possible non-deterministic behaviour of non-atomic methods allowed by the JAVA CARD specification. Our implementation would still have to rely on the `arrayCopyNonAtomic` method to provide unconditional updates of the counter, and, at the same time, provide a mechanism to neutralise possible faults caused by card tears. Our experiments with the non-atomic methods suggest that once a call to a non-atomic method is completed, the data that was modified by the method is stable, that is, a card tear does not affect the already modified value. Thus, the main idea to implement a secure try counter is to keep three copies of the counter and update them one after another. If a card tear occurs we can assume that two of the three copies will have a correct value–either the old one or the new one. After the card tear, by analysing the differences in the three copies we can establish what the correct value of the try counter should be. Each time a read or a write operation on the try counter is invoked, we first perform such analysis of the three copies to see if they represent a stable state (all three copies are equal) of the try counter. If not, we restore the values of the three copies

back to a stable state. This way we correct the faults that were caused by a card tear. The actual code implementing this scheme is given below:

```
public class TryCounter {

  private byte[] temp;
  private byte max;
  private byte[] c1;
  private byte[] c2;
  private byte[] c3;

  public TryCounter(byte max) {
    temp =
      JCSystem.makeTransientByteArray(1, JCSystem.CLEAR_ON_RESET);
    c1 = new byte[1];
    c2 = new byte[1];
    c3 = new byte[1];
    this.max = max;
    setNA(max);
  }

  private void setNA(byte value) {
    temp[0] = value;
    Util.arrayCopyNonAtomic(temp, 0, c1, 0, 1);
    Util.arrayCopyNonAtomic(temp, 0, c2, 0, 1);
    Util.arrayCopyNonAtomic(temp, 0, c3, 0, 1);
  }

  private void restore() {
    if(c1[0] == c2[0] && c2[0] == c3[0]) {
      return;
    }
    if(c2[0] == c3[0]) {
      // Tear occurred during the update of the first copy:
      temp[0] = c3[0];
      Util.arrayCopyNonAtomic(temp, 0, c1, 0, 1);
      return;
    }
    if(c1[0] == c2[0]) {
      // Tear occurred during the update of the last copy:
      temp[0] = c1[0];
      Util.arrayCopyNonAtomic(temp, 0, c3, 0, 1);
      return;
    }
    // All three different, tear occurred during the
    // update of the second copy, the third copy should
```

```
    // be considered to be the stable one:
    temp[0] = c3[0];
    Util.arrayCopyNonAtomic(temp, 0, c2, 0, 1);
    Util.arrayCopyNonAtomic(temp, 0, c1, 0, 1);
  }

  public byte getValue() {
    restore();
    // Double check, if something is amiss, return 0
    if (c1[0] == c2[0] && c2[0] == c3[0])
      return c1[0];
    return 0;
  }

  public void decrease() {
    restore();
    // If not already 0, decrease the counter by 0
    if(c1[0] == (byte)0) {
      return;
    }
    setNA(c1[0]-1);
  }

  public void max() {
    restore();
    setNA(max);
  }
}
```

This should provide a secure implementation of a try counter, which can then be used in a secure, non-native, implementation of the `OwnerPIN` class. We changed the insecure reference implementation of `OwnerPIN` discussed in Section 5 to use this implementation of a try counter, and then repeated our card tear attacks on this new implementation of `OwnerPIN`. As we expected, with the new try counter implementation, the `OwnerPIN` could not be exploited.

Of course, such experiments attacking an implementation can only reveal the presence of security vulnerabilities, but cannot prove their absence. How we can guarantee the absence of any security vulnerabilities caused by card tearing in a supposedly defensive implementation such as the `TryCounter` class above is the topic of the next section. For this we turn to formal methods, more specifically, model checking.

6.2 Verifying the Absence of Faults

The correctness of the `TryCounter` class above, and its security even in the presence of card tears, is quite a subtle issue. For instance, the order of updates

to the three registers in the `restore` method is crucial. If the order would be changed, repeated card tears could manipulate the three copies in such a way that an inconsistent value of the try counter would be reached. Performing the card tear tests repeatedly on this implementation did not reveal any problems of this kind—a call to `getValue` always returned either the value the counter had before or the value it was supposed to have after the interrupted update operation.

However, such repeated random tests do not prove that this is always going to be the case. What we would like to establish is that *all* possible card tears during calls to `TryCounter` methods will never cause the try counter value to be inconsistent. To prove this formally we turn to model checking. We used the model checking tool Uppaal[6] [4], simply because we have experience with this tool—other model checking tools could be used too.

State Machine. The first necessary thing to perform model checking is to define a state machine that models the behaviour of the `TryCounter` class.

This state machine not only models the normal behaviour, but also the behaviour in case a card tear occurs. This behaviour includes the simple interruption of execution and the subsequent clearing of transient data that happens when poser is lost. It also includes the behaviour in case a card tear happens during a call of `arrayCopyNonAtomic`, which will not only interrupts the execution and clear all transient data, but will also set the contents of the array being modified to (possibly) random values.

The initial state of the model represents the "entry point" of the `TryCounter` class—in this state any of the three public methods of the class can be called. From this initial state there are three outgoing sequences of transitions, one for each of the three public methods. (In fact, because each of the public methods start with a call to `restore()`, the transitions modelling the body of `restore()` can be joined.) At any point in the execution a card tear may occur, which is modelled by transitions back to the initial state. Here we treat the individual Java Card byte-code instructions as atomic, i.e. every byte-code instruction is a transition between two program points, and card tears can happen in the state before or after any individual byte-code instruction. At any program point where `arrayCopyNonAtomic` is invoked, there is not only an outgoing transition to the next program point that models the successful completion of the method call, but also an outgoing transition back to the initial state that models the interruption of the method call, and which resets the array contents to random values.

So the state machine models all possible lifetimes of a `TryCounter` object: it models all possible sequences of method invocations, with all possible interruptions by card tears. The resulting model is highly non-deterministic, but it is finite, because the state of a try counter object—consisting of only five instance fields—is finite.

[6] `http://www.uppaal.com`

Properties. We specified two properties about the state machine:

- in the final "stable" state the values of the three copies of the try counter are always the same;
- in the final state, the stable value of the try counter is always equal either to the old stable value of the counter, recorded just after a call to `restore`, or the new value that it was supposed to be updated to—the old value minus one, or the maximum try counter value.

Model Checking. We constructed the state machine and specified the two properties in the Uppaal model checker [4]. Uppaal has a graphical interface for the editing state machines, making it very easy to use. Unfortunately, the state machine for `TryCounter` is too large to be included as a picture in the report here. Both properties were quickly verified. That provides a proof that the implementation of `TryCounter` is indeed secure with respect to faults caused by card tears and any unpredictable behaviour this may cause during invocations of non-atomic methods.

Of course, the whole formalisation and the proof are only valid under the assumptions we made about the possible effect of card tears on the execution. Here we took into account the official JAVA CARD but also all behaviour of cards that we observed in our experiments that violated this specification. So the results are valid for any card that correctly implements the JAVA CARD specification and all the cards that we actually tested, assuming that these cards do not have bugs that our testing did not reveal. For other cards that we did not test, and which may violate the JAVA CARD specification in other ways, the behaviour of non-atomic methods in the context of card tears may differ substantially and possibly invalidate this correctness proof.

Verification of TryCounter with KeY. The correctness of `TryCounter` assuming there are no card tears can also be verified with the KeY system. In fact, we gave full behavioural specification of the `TryCounter` class in JML and verified it with the KeY system. For example, we specified and proved that the complete execution, without intermediate card tears, of the `restore` method indeed results in the three copies of the try counter equal to each other. This correctness result proved with KeY is subsumed by the correctness result proved with the model checker.

7 Summary and Discussion

The main conclusions of our work are:

- The transaction mechanism is one of the most complex parts of the JAVA CARD technology. The ambiguities and under-specifications in the official specification show that it is difficult to formalise. The behaviour of the actual smart cards shows that it is also quite difficult to implement.

- Despite the under-specifications allowing "liberal" implementation some cards still have faulty (non-compliant) implementation of non-atomic methods. In particular, some cards also produce unpredictable results if non-atomic methods are interrupted while no transaction is in progress.
- Extreme care has to be taken when using non-atomic methods for security-sensitive operations. However, despite the possible faults it is still possible to utilise non-atomic methods by defensive programming.
- Finally, we showed how two different formal verification techniques—program verification and model checking—can be used to provide rigorous proof of absence of bugs, even in the presence of possible faults.

These results bring up a number of very interesting discussion points. First of all, the usefulness of non-atomic methods should be questioned. As far as we can see the existence of non-atomic methods is justified by the following three goals:

- *A possibility to by-pass the transaction mechanism.* For security reasons one sometimes needs to by-pass the transaction mechanism. However, given the unpredictable results when non-atomic methods are interrupted by card tears, it is dangerous to rely on these methods to do so. Our `TryCounter` implementation demonstrates that non-atomic methods can be used to safely by-pass the transaction, and moreover, that we can formally provide assurance of its correctness. However, this relies on non-trivial defensive programming techniques. Also, we cannot give any formal assurance of correctness for cards that have bugs that we have not discovered.
- *A fast way to copy transient data.* Since transient data is always reset after a card tear, it can be safely modified with a non-atomic method.
- *Finally, a fast way to copy persistent data outside of a transaction, in cases where the data can be left in a partially modified state.* Here, only if one can allow unpredictable contents of the modified array, the method can be used. If one would like to ensure that the array is "partially modified" (that is, the property suggested by the official specification, but not satisfied by some of the cards), the non-atomic method is not safe on all cards. So it seems the only option is to use the atomic version of the array copy method, `arrayCopy` from the class `Util`. The method `arrayCopy`, however, will not work for copying large amounts of data; because `arrayCopy` uses the transaction mechanism, the transaction commit buffer can be easily exhausted. We tried to copy an array of 1024 bytes with `arrayCopy`—a transaction exception was reported. Thus, to copy a large array one *has* to use either a non-atomic method (may result in unacceptable faults), or simply use a `for` loop (but this will be considerably slower than using a native method).

The overall conclusion would be that non-atomic methods can only be used safely on transient data, when applying these methods to persistent data extreme care has to be taken. In particular, using non-atomic methods to by-pass the transaction mechanism is possible but defensive programming techniques have to be used to ensure security in the presence of card tears.

The second issue is JAVA CARD interoperability. Disregarding the fact that some of the cards fail to actually meet the official specification, the fact is that cards tend to exhibit slightly different behaviour in the context of transactions. Apart from the results of the non-atomic methods test, we also discovered the different behaviour with respect to 'maxing' try counter in the `OwnerPIN` implementation. Some of the reasons for this situation lay in the official JAVA CARD specification. Firstly, it contains some ambiguities leaving freedom of interpretation to the implementors. Secondly, if not ambiguous, the specification is deliberately under-specified—it explicitly allows the card behaviour to differ. The statement about the possible unpredictable result of calling a non-atomic method inside a transaction has been introduced to the JAVA CARD specification with the release of the 2.2 version of the specification. Although we are not certain, we suspect that this was done because some of the cards were discovered (as we did) to exhibit such an unpredictable behaviour. This would suggest that the behaviour of the cards influences the specification, and not the other way round. We leave this last statement without any more comments.

Finally, we demonstrated how formal verification techniques can be used to guarantee security in the presence of faults, providing a higher level of assurance than testing can. Here we used two techniques: *program verification* with the KeY tool and *model checking* with Uppaal.

Program verification using the KeY tool can be used to verify properties of code under the assumption that no card tears happen. If we could assume there was no unpredictable behaviour of non-atomic methods, the KeY system could be used to also reason about JAVA CARD programs in the presence of card tears. (Without this assumption, reasoning in the presence of card tears is in principle possible with KeY, but it would require further work.) Model checking using Uppaal, on the other hand, can also consider the non-deterministic behaviour of cards if card tears occur.

A drawback of the model checking approach was that we had to rely on a model constructed by hard, which can be error-prone. Indeed, we did not get it right at the first try. (In principle, it is possible to automate this; indeed, the Bogor tool [19] can be used to model-check JAVA programs, but this tool is aimed at multi-threaded JAVA programs and is unaware of card tears as a source of non-determinism in JAVA CARD.) Another drawback of the model checking approach is that because of the state explosion problem it can only cope with very small programs. Program verification tools can cope with programs the size of typical JAVA CARD applets, model checkers cannot unless tricks are used to reduce the state space. Still, crucial program components such as implementation of an `OwnerPIN` class are within the reach of model checkers. An interesting research question for future work is to see if the model-checking approach can be used to include other sources of fault injections, to check, for instance, if implementations of API classes such as `OwnerPIN` are resistant to other fault injection attacks besides card tears.

Program verification with the KeY tool does not require manual construction of a model, but works directly on the JAVA CARD source code. Properties can also

be specified at the level of source code, in the specification language JML [14]. Because of this, users may find such a tool easier to use than a model checker, despite the fact that model checking has the advantage of being fully automated, and program verification requires more user interaction.

Program verification tools for JAVA have been used for JAVA CARD applets in the past, e.g. see [5, 17], and other program verification tools for JAVA than KeY have been used for this, for instance ESC/JAVA [9] and its successor tool ESC/JAVA2 [8], JACK [6], or Krakatoa [15]. However, none of these other tools model the special features of JAVA CARD such as the transaction mechanism and the distinction between transient and persistent memory, although for the Krakatoa tool a work to include support for the transaction mechanism has been recently reported [16].

The final question is whether the official JAVA CARD specification could or should be made stricter, by removing the ambiguity discussed in Section 3 and, more importantly, not including the deliberate under-specification for the non-atomic methods in the event of a card tear. One of the cards tested demonstrates that it is technically possible to meet such a stricter specification, where in particular the result of a card tear during calls to non-atomic methods is not unpredictable. Such a stricter specification would improve portability of applications. Also, tricky defensive programming techniques would no longer be necessary to prevent faults injection attacks by card tearing, and (formal) verification of security properties in the event of card tears would be much easier.

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.

This work is supported by the research program Sentinels (`http://www.sentinels.nl`). Sentinels is financed by the Technology Foundation STW, the Netherlands Organisation for Scientific Research (NWO), and the Dutch Ministry of Economic Affairs.

The work of Erik Poll is funded in part by the Information Society Technologies programme of the European Commission, Future and Emerging Technologies, in the MOBIUS project (IST-2005-015905).

References

1. Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Richard Bubel, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, Andreas Roth, Steffen Schlager, and Peter H. Schmitt. The KeY tool. *Software and Systems Modeling*, 4(1):32–54, February 2005.
2. Bernhard Beckert. A dynamic logic for the formal verification of JAVA CARD programs. In I. Attali and T. Jensen, editors, *JAVA on Smart Cards: Programming and Security. Revised Papers, JAVA CARD 2000, International Workshop, Cannes, France*, volume 2041 of *LNCS*, pages 6–24. Springer, 2001.

3. Bernhard Beckert and Wojciech Mostowski. A program logic for handling JAVA CARD's transaction mechanism. In Mauro Pezzè, editor, *Proceedings, Fundamental Approaches to Software Engineering (FASE) Conference 2003, Warsaw, Poland*, volume 2621 of *LNCS*, pages 246–260. Springer, April 2003.
4. Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on UPPAAL. In Marco Bernardo and Flavio Corradini, editors, *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, number 3185 in LNCS, pages 200–236. Springer, September 2004.
5. Cees-Bart Breunesse, Nestor Cataño, Marieke Huisman, and Bart Jacobs. Formal methods for smart cards: an experience report. *Science of Computer Programming*, 55:53–80, 2005.
6. Lilian Burdy, Antoine Requet, and Jean-Louis Lanet. JAVA applet correctness: A developer-oriented approach. In *Proceedings, Formal Methods Europe 2003*, volume 2805 of *LNCS*, pages 422–439. Springer, 2003.
7. Zhiqun Chen. *JAVA CARD Technology for Smart Cards: Architecture and Programmer's Guide*. JAVA Series. Addison-Wesley, June 2000.
8. David R. Cok and Joseph R. Kiniry. ESC/JAVA2: Uniting ESC/JAVA and JML. Technical report, University of Nijmegen, 2004. NIII Technical Report NIII-R0413.
9. Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for JAVA. In *Proceedings, ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation, Berlin*, pages 234–245. ACM Press, 2002.
10. Reiner Hähnle and Wojciech Mostowski. Verification of safety properties in the presence of transactions. In Gilles Barthe and Marieke Huisman, editors, *Proceedings, Construction and Analysis of Safe, Secure and Interoperable Smart devices (CASSIS'04) Workshop*, volume 3362 of *LNCS*, pages 151–171. Springer, 2005.
11. Engelbert Hubbers and Erik Poll. Reasoning about card tears and transactions in JAVA CARD. In *Fundamental Approaches to Software Engineering (FASE'2004), Barcelona, Spain*, volume 2984 of *LNCS*, pages 114–128. Springer, 2004.
12. Engelbert Hubbers and Erik Poll. Transactions and non-atomic API calls in JAVA CARD: Specification ambiguity and strange implementation behaviours. Deptartment of Computer Science NIII-R0438, Radboud University Nijmegen, 2004.
13. Bart Jacobs and Erik Poll. JAVA program verification at Nijmegen: Developments and perspective. In *Software Security – Theories and Systems: Second Mext-NSF-JSPS International Symposium, ISSS 2003, Tokyo, Japan, November 4–6, 2003. Revised Papers*, volume 3233 of *LNCS*, pages 134–153. Springer, 2003.
14. Gary T. Leavens, Albert L. Baker, and Clyde Ruby. *JML: A Notation for Detailed Design*. Kluwer Academic Publishers, 1999.
15. Claude Marché, Christine Paulin-Mohring, and Xavier Urbain. The KRAKATOA tool for certification of JAVA/JAVA CARD programs annotated in JML. *Journal of Logic and Algebraic Programming*, 58(1–2):89–106, 2004.
16. Claude Marché and Nicolas Rousset. Verification of JAVA CARD applets behavior with respect to transactions and card tears. In *Proceedings, Software Engineering and Formal Methods (SEFM), Pune, India*. IEEE CS Press, 2006. To appear.
17. Wojciech Mostowski. Formalisation and verification of JAVA CARD security properties in Dynamic Logic. In Maura Cerioli, editor, *Proceedings, Fundamental Approaches to Software Engineering (FASE) Conference 2005, Edinburgh, Scotland*, volume 3442 of *LNCS*, pages 357–371. Springer, April 2005.

18. Wojciech Mostowski. Formal reasoning about non-atomic Java Card methods in Dynamic Logic. In Tobias Nipkow and Jayadev Misra, editors, *Proceedings, Formal Methods (FM) 2006*, LNCS. Springer, 2006. To appear.
19. Robby, Edwin Rodríguez, Matthew Dwyer, and John Hatcliff. Checking strong specifications using an extensible software model checking framework. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004*, volume 2988 of *Lecture Notes in Computer Science*, pages 404–420. Springer, 2004.
20. Sun Microsystems, Inc., `http://www.sun.com`. *Java Card 2.1.1 Runtime Environment Specification*, May 2000.
21. Sun Microsystems, Inc., `http://www.sun.com`. *Java Card 2.2.2 API Specification*, March 2006.
22. Sun Microsystems, Inc., `http://www.sun.com`. *Java Card 2.2.2 Runtime Environment Specification*, March 2006.
23. Jos Warmer and Anneke Kleppe. *The Object Constraint Language, Second Edition: Getting Your Models Ready for MDA*. Object Technology Series. Addison-Wesley, Reading/MA, 2003.