Malicious Code on Java Card Smartcards: Attacks and Countermeasures

Wojciech Mostowski and Erik Poll

Digital Security (DS) group, Department of Computing Science
Radboud University Nijmegen, The Netherlands
{woj,erikpoll}@cs.ru.nl

Abstract. When it comes to security, an interesting difference between Java Card and regular Java is the absence of an on-card bytecode verifier on most Java Cards. In principle this opens up the possibility of malicious, ill-typed code as an avenue of attack, though the Java Card platform offers some protection against this, notably by code signing.
This paper gives an extensive overview of vulnerabilities and possible runtime countermeasures against ill-typed code, and describes results of experiments with attacking actual Java Cards currently on the market with malicious code.

1 Overview

A huge security advantage of type safe language such as Java is that the low level memory vulnerabilities, which plague C/C++ code in the form of buffer overflows, are in principle ruled out. Also, it allows us to make guarantees about the behaviour of one piece of code, without reviewing or even knowing all the other pieces of code that may be running on the same machine.

However, on Java Card smartcards [9] an on-card bytecode verifier (BCV) is only optional, and indeed most cards do not include one. This means that malicious, ill-typed code is a possible avenue of attack.

Of course, the Java Card platform offers measures to protect against this, most notably by restricting installation of applets by means of digital signatures – or disabling it completely. Still, even if most Java Card smartcards that are deployed rely on these measures to avoid malicious code, it remains an interesting question how vulnerable Java Card applications are to malicious code. Firstly, the question is highly relevant for security evaluations of code: can we evaluate the code of one applet without looking at other applets that are on the card? Secondly, the defence mechanisms of the Java Card platform are not so easy to understand; for instance, there is the firewall as an extra line of defence, but does that offer any additional protection against ill-typed code, and can it compensate for the lack of BCV? And given the choice between cards with and without BCV, are there good reasons to choose for one over the other? (As we will show, cards with on-card BCV may still be vulnerable to ill-typed code!)

In this paper we take a serious look at the vulnerability of the Java Card platform against malicious, ill-typed code. We consider the various ways to get

ill-typed code on a smartcard, and various ways in which one applet could try to do damage to another applet or the platform, and the countermeasures the platform might deploy.

There is surprisingly little literature on these topics. The various defences of the Java Card platform are only given rather superficial discussion in [6, Chapter 8]. The only paper we know that discusses type flaw attacks on a Java Card smartcard is [11].

We have experimented with attacks on eight different cards from four manufacturers, implementing Java Card versions 2.1.1, 2.2, or 2.2.1. We will refer to these cards as `A_211`, `A_221`, `B_211`, `B_22`, `B_221`, `C_211A`, `C_211B`, and `D_211`. The first letter indicates the manufacturer, the numbers indicate which Java Card version the card provides. Based on the outcome of the experiments, we can make some educated guesses on which of the countermeasures the cards actually implement.

The outline of the rest of this paper is as follows. Sect. 2 briefly reviews the different lines of defence on the Java Card platform, including bytecode verification and the applet firewall. The first step in any attack involving ill-typed code is getting ill-typed code installed on the smartcard. More precisely, we want to somehow create type confusion, or break the type system, by having two pieces of code treat (a reference to) the *same* piece of physical memory as having different, incompatible types. Sect. 3 discusses the various ways to create type confusion and the success we had with these methods on the various cards. Sect. 4 then discusses experiments with concrete attacks scenarios.

Sect. 5 discusses the various runtime countermeasures the platform could implement against such attacks, some of which we ran into in the course of our experiments. Finally, Sect. 6 summarises our results and discusses the implications.

2 Defences

In this section we briefly describe and compare the protection provided by the various protection mechanisms on a Java Card platform.

2.1 Bytecode Verification

Bytecode verification of Java (Card) code guarantees type correctness of code, which in turn guarantees memory safety. For the normal Java platform, code is subjected to bytecode verification at load time. For Java Cards, which do not support dynamic class loading, bytecode verification can be performed at installation time (when an applet is installed on the smartcard). However, most Java Card smartcards do not have an on-card BCV, and check a digital signature of a third party who is trusted to have performed bytecode verification off-card.

Note that even if bytecode is statically verified, some checks will always have to be done dynamically, namely checks for non-nullness, array bounds, and downcasts. For Java Card, the applet firewall will also require runtime checks.

Although typically bytecode verification is done statically, it is also possible to do type checking dynamically, i.e. at runtime, by a so-called defensive virtual machine. This requires keeping track of typing information at runtime and performing type checks prior to the execution of every bytecode instruction. Clearly, this has an overhead both in execution time and in memory usage. However, to check downcasts the VM already has to record runtime types of objects anyway.

As we will see later, our experiments show that some Java Cards do a form of runtime type checking, and this then offers an excellent protection against ill-typed code.

2.2 Applet Firewall

The applet firewall is an additional defence mechanism implemented on all Java Cards. The firewall performs checks at runtime to prevent applets from accessing (reading or writing) data of other applets (of applets in a different security context, to be precise). For every object its context is recorded, and for any field or method access it is checked if it is allowed. In a nutshell, applets are only allowed to access in their own context, but the Java Card Runtime Environment (JCRE) has the right to access anything. In UNIX terminology, the JCRE executes in root-mode, and some of the Java Card API calls, which are executed in JCRE context, are 'setuid root'.

Defence mechanisms can be *complimentary*, each providing different guarantees that the other cannot, or *defence in depth*, each providing the same guarantees, so that one can provide a back-up in case the other fails. As defence mechanisms, the firewall and bytecode verification are primarily complimentary. The firewall provides additional guarantees that bytecode verification does not: a carelessly coded applet might expose some of its data fields by declaring them public, allowing other applets to read or modify them.[1] Bytecode verification cannot protect against this, but the firewall will. The firewall provides a strong guarantee that an applet cannot be influenced by the behaviour of other (well-typed!) applets, unless it explicitly exposes functionality via a so-called Shareable Interface Object.

The firewall is not guaranteed to offer protection against ill-typed applets. Still, for certain attack scenarios, the firewall does provide a useful second line of defence. If a malicious applet manages to get hold of a 'foreign' reference to an object belonging to another applet by breaking the type system, its access to that object reference is still subject to runtime firewall checks, which may prevent any harm.

Breaking the Firewall The firewall as specified in [9] is quite complicated. This means that there is a real chance of implementation bugs, or unclarities in the specs, which might lead to security flaws. We investigated the firewall

[1] Indeed, security coding guidelines for Java such as [6, Chapter 7] or `http://www.dwheeler.com/secure-programs/Secure-Programs-HOWTO/java.html` warn against the use of public, package, and protected (!) visibility.

specification and thoroughly tested cards for any flaws. Details are described in a separate technical report [8]. We did find some minor deviations from the official specification on some cards, but most of them 'safe', in the sense that the cards were more restrictive than the specification demanded. The only 'unsafe' violation of the specifications we found was that card `A_221` ignores the restriction that access via a shareable interface should not be allowed when an applet is active on another logical channel. This could lead to security problems in particular applications that use shareable interfaces. The tests in [8] only consider well-typed code. However, a weak firewall implementation can be broken with ill-typed code, as we will discuss in Sect. 4.2.4.

3 Getting Ill-Typed Code on Cards

We know four ways to get ill-typed code onto a smartcard: (i) CAP file manipulation, (ii) abusing shareable interface objects, (iii) abusing the transaction mechanism, and (iv) fault injection. Note that an on-card BCV is only guaranteed to protect against the first way to break type safety (assuming the BCV is correct, of course). These methods are discussed in more detail below, and for the first three we discuss whether they work on the cards we experimented with.

3.1 CAP File Manipulation

The simplest way to get ill-typed code running on a card is to edit a CAP (Converted APplet) file to introduce a type flaw in the bytecode and install it on the card. Of course, this will only work for cards without on-card BCV and for unsigned CAP files. One can make such edits in the CAP file or – which is easier – in the more readable JCA (Java Card Assembly) files. For example, to treat a `byte` array as a `short` array, it is enough to change a `baload` (byte load) opcode into a `saload` (short load). Such a misinterpreted array type can potentially lead to accessing other applets' memory as we explain in Section 4. To further simplify things, instead of editing JCA or CAP files, one could use some tool for this; ST Microelectronics have developed such a tool, which they kindly let us use.

CAP file editing gives endless possibilities to produce type confusion in the code, including type confusion between references and values of primitive types, which in turn allows C-like pointer arithmetic.

Experiments As expected, all the cards without on-card BCV installed ill-typed CAP files without problems. Apart from an on-card BCV, simply prohibiting applet loading – common practice on smartcards in the field – or at least controlling applet loading with digital signatures are of course ways to prevent against manipulated CAP files.

3.2 Abusing Shareable Interface Objects

The shareable interface mechanism of Java Card can be used to create type confusion between applets without any direct editing of CAP files, as first suggested in [11].

Shareable interfaces allow communication between applets (or between security contexts, to be precise): references to instances of shareable interfaces can be legally used across the applet firewall. To use this to create type confusion, the trick is to let two applets communicate via a shareable interface, but to compile and generate CAP files for the applets using different definitions of the shareable interface. This is possible because the applets are compiled and loaded separately.

For example, suppose we have a server applet exposing a shareable interface `MyInterface` and a second client applet using this interface. If we produce the CAP file for the server applet using the following interface definition

```
public interface MyInterface extends Shareable {
  void accessArray(short[] array); } // Server assumes short[]
```

and the CAP file for the client applet using

```
public interface MyInterface extends Shareable {
  void accessArray(byte[] array); } // Client assumes byte[]
```

then we can make the server applet treat a `byte` array as a `short` array.

One last thing to take care of in this scenario is to circumvent the applet firewall mechanism. Since the server and client applet reside in different contexts, the server does not have the right to access the array it gets from the client. Hence, to make this work the server has to first send its own `byte` array reference to the client and then the client has to send it back to the server through the ill-typed interface. This way the server can run malicious code on its own (in terms of context ownership) data. Now, the shareable interface definition for the server will for instance include

```
  void accessArray(short[] array); // Server assumes short[]
  byte[] giveArray();              // Server gives its array to client
```

whereas the one for the client includes

```
  void accessArray(byte[] array); // Client assumes byte[]
  byte[] giveArray();             // This array from server is sent back
                                  // to the server with accessArray(...)
```

Obviously, this scenario is not limited to confusing `byte` arrays with `short` arrays. Virtually any two types can be confused this way.

We should point out that the client and server applet usually need to be aware of each other and actively cooperate to cause an exploitable type unsoundness. So they both have to be malicious. To the best of our analysis it is not really possible to type-attack an 'unaware' server which exports a shareable interface, by crafting a malicious client applet, or vice versa.

Experiments This method to break the type system worked on all cards without BCV, with the exception of `D_211`. Card `D_211`, without on-card BCV, refused to load any code that uses shareable interfaces – for reasons still unclear to us.

Both cards with on-card BCV, `C_211A` and `C_211B`, also refuse to install code that uses shareable interfaces, but that is understandable. An on-card BCV may

have a hard time spotting type flaws caused by the use of incompatible interfaces definition, because just performing bytecode verification of an individual applet will not reveal that anything is wrong. So cards `C_211A` and `C_211B` resort to a simpler and more extreme approach: they simply refuse to load any code that use shareable interfaces. This clearly avoids the whole issue with type checking such code. (Strictly speaking, one can argue that these cards do not implement the Java Card standard correctly, as there is no mention in the Java Card specification of shareable interfaces being an optional feature.)

3.3 Abusing The Transaction Mechanism

The Java Card transaction mechanism, defined in [9], is probably the trickiest aspect of the Java Card platform. The mechanism has been the subject of investigation in several papers [1, 5], and [4] demonstrated it as a possible source of fault injections on one card. The transaction mechanism allows multiple bytecode instructions to be turned into an atomic operation, offering a roll-back mechanism in case the operation is aborted, which can happen by a card tear or an invocation of the API method `JCSystem.abortTransaction`. The roll-back mechanism should also deallocate any objects allocated during an aborted transaction, and reset references to such objects to `null` [9, Sect. 7.6.3].

As pointed out to us by Marc Witteman, it is this last aspect which can be abused to create type confusion: if such references are spread around, by assignments to instance fields and local variables, it becomes difficult for the transaction mechanism to keep track of which references should be nulled out. (This problem is similar to reference tracking for garbage collection, which is normally not supported on Java Cards.) For example, consider the following program:

```
short[] array1, array2; // instance fields
...
  short[] localArray = null; // local variable
  JCSystem.beginTransaction();
    array1 = new short[1];
    array2 = localArray = array1;
  JCSystem.abortTransaction();
```

Just before the transaction is aborted, the three variables `array1`, `array2`, and `localArray` will all refer to the same `short` array created within the transaction. After the call to `abortTransaction`, this array will have been deallocated and all three variables should be reset to `null`.

However, buggy implementations of the transaction mechanism on some cards keep the reference in `localArray` and reset only `array1` and `array2`. On top of this, the new object allocation that happens after the abort method reuses the reference that was just (seemingly) freed. Thus the following code:

```
short[] arrayS; // instance field
byte[]  arrayB; // instance field
...
```

```
short[] localArray = null; // local variable
JCSystem.beginTransaction();
  arrayS = new short[1]; localArray = arrayS;
JCSystem.abortTransaction();
arrayB = new byte[10]; // arrayB gets the same reference as arrayS
                       // used to have, this can be tested
if((Object)arrayB == (Object)localArray) ... // this is true!
```

produces two variables of incompatible types, `arrayB` of type `byte[]` and `localArray` of type `short[]`, that hold the same reference, so we have ill-typed code. Again, this trick is not limited to array types.

The root cause of this problem is the subtle 'feature' of the transaction mechanism that stack-allocated variables, such as `localArray` in the example above, are *not* subject to roll-back in the event of a programmatically aborted transaction (i.e. a call to `JCSystem.abortTransaction`). Apparently this potential for trouble has been noticed, as the JCRE specification [9, Sect. 7.6.3] explicitly allows a card to mute in the event of a programmatic abort after objects have been created during the transaction.

Experiments Four cards (`B_211`, `B_221`, `C_211A`, `D_211`) have a buggy transaction mechanism implementation that allows the type system to be broken in the way described above. Note that an on-card BCV will *not* prevent this attack. Indeed, one of the cards with an on-card BCV, `C_211A`, is vulnerable to this attack.

The obvious countermeasure against this attack is to correctly implement the clean-up operation for aborted transactions. However, only one of our test cards (`B_22`) managed to perform a full clean-up correctly.

Another countermeasure against this attack is for cards to mute when a transaction during which objects have been created is programmatically aborted. As mentioned above, this is allowed by the JCRE specifications. Three cards we tested (`A_211`, `A_221`, `C_211B`) implement this option.

3.4 Fault Injections

Finally, fault injections, e.g. by light manipulations, could in theory be used to change the bytecode installed on a card and introduce type flaws.

Fault injections do not provide a very controlled way to change memory, so the chance of getting an exploitable type flaw is likely to be small. However, following the ideas described in [3], for specially tailored code a fault injection can be expected to produce an exploitable type flaw with a relatively high chance.

We did not carry out any experiments with this, since we do not have the hardware to carry out fault injections.

4 Type Attacks on Java Cards

Using the various methods discussed in the previous section, we were able to install ill-typed code on all but one of the smartcards we had (namely card `C_211B`). We then experimented with various attacks on these cards.

One idea was to exploit type confusion between primitive arrays of different types, a `byte` array with a `short` array, to try and access arrays beyond the array bounds. Another basic idea was to exploit type confusion between an array and a object that was not an array, where there are several opportunities for mischief, such as redefining an array's length – reportedly successful on some cards [11] – or writing object references to perform pointer arithmetic or spoof references. Whether confusion between arrays and other objects can be exploited depends on the exact representation of objects in memory.

4.1 Accessing a Byte Array as a Short Array [byte as short array]

The first attack is to try to convince the applet to treat a `byte` array as a `short` array. In theory this would allow one to read (or write) twice the size of the original `byte` array. For instance, accessing a `byte` array of length 10 as a `short` array size might allow us to access 10 shorts, i.e. 20 bytes, with the last 10 bytes possibly belonging to another applet.

We considered three kinds of byte arrays: global arrays (i.e. the APDU buffer), persistent context-owned arrays, and transient context-owned arrays. There was no difference in behaviour between these different kinds of arrays, i.e. each card gives the same result for all three array types. However, different cards did exhibit different behaviour, as described below.

Cards `C_211A` and `D_211` gave us the result we were hoping for, as attackers. It was possible to read beyond the original array bound. In particular, we managed to access large parts of the CAP file that was later installed on the card. This is clearly dangerous, as it means an applet could read and modify code or data of another applet that was installed later. NB `C_211A` is the card *with* on-card BCV where the buggy transaction mechanism allowed us to run ill-typed code. This highlights the danger of putting all one's trust in an on-card BCV!

Cards from manufacturer B did not let us read a single value from an ill-typed array. Cards `B_211` and `B_221` muted the current card session whenever we tried this, and `B_22` returned a response APDU with status word `6F48`. This suggests that these cards perform runtime type checking (at least enough to detect this particular type confusion). Indeed, all attacks we tried on `B_211` and `B_221` were ineffective in that they always gave this same result, i.e. the cards muted whenever an ill-typed instruction was executed. For card `B_22` some attacks did give more interesting results.

Results on cards from manufacturer A were hardest to understand. Card `A_221` allowed us to run the ill-typed code. However, it does not let us read data outside the original array bounds. What happens is that one byte value is treated as one short value (exactly as if bytes were in fact implemented as shorts in the underlying hardware). For positive byte values each value is prepended with a `00`, for negative values with `FF`:

```
Read as byte[]  00   01   02   03   04   ... 80   81   ...
Read as short[] 0000 0001 0002 0003 0004 ... FF80 FF81 ...
```

Card A_211 allowed us to run the ill-typed code and reads two subsequent bytes as one short value:

```
Read as byte[]  00 01 02 03 04 05 06 07 ...
Read as short[] 0001  0203  0405  0607 ...
```

However, it was not possible to go outside of the range of the original byte array: even though the presumed short array reports to be of size n, it is only possible to access the first $n/2$ elements, allowing access to the original n bytes. Attempts to access array elements beyond this yielded an `ArrayIndexOutOfBoundsException`.

What appears to be happening on A_211 is that array bounds checks are done in terms of the physical memory size of the arrays (in bytes), not in terms of the logical size of the arrays (in number of elements). We will call this *physical bounds checking*.

4.2 Accessing an Object as an Array [object as array]

Instead of confusing two arrays of different type, one can more generally try to confuse an arbitrary object with an array. This opens the following attack possibilities.

4.2.1 Fabricating Arrays Witteman [11] describes a type attack which exploits type confusion between an array and an object of the following class `Fake`:

```
public class Fake { short len = (short)0x7FFF; }
```

The attack relies on a particular representation of arrays and objects in memory: for the attack to succeed, the length field of an array has to be stored at the same offset in physical memory as the `len` field of a `Fake` object. If we can then trick the VM in treating a `Fake` object as an array, then the length of that array would be `0x7FFF`, giving access to 32K of memory. The fact that we can access the `len` field through the object reference could allow us to set the array length to an arbitrary value.

Although setting the length of the array was not possible as such on the cards we tested, this attack gave us interesting results nevertheless.

As before, cards B_211 and B_221 refused to execute the ill-typed code, as in all other attack scenarios. Card A_211 also refused to execute the code, returning the 6F00 status word. Apparently A_211 has some runtime checks that prevent type confusion between objects and arrays.

On the cards where running our exploit code was possible (A_221, B_22, C_211A, D_211), the object representation in memory prevents us from manipulating the array length. Still, we noticed two different behaviours. For cards A_221 and B_22, the length of the "confused" array indicates the number of fields in the object, i.e. an instance of the `Fake` class gives an array of length 1 containing the element `0x7FFF`. For the two other cards, C_211A and D_211, the length of the confused array has some apparently arbitrary value: the length

is not the number of instance fields, but it probably represents some internal object data. For C_211A this value is large enough (actually negative when the length is interpreted as a signed short) to freely access any forward memory location on the card.

A slight modification of this attack allows us to read and write object references directly as we describe next.

4.2.2 Direct Object Data Access The results of the previous attack suggests that it would be possible to treat object fields as array elements on cards A_221, B_22, C_211A, and D_211. Reference fields could then be read or written as numerical values, opening the door to pointer arithmetic.

This is indeed possible for all these cards. For example, an instance of this class

```
public class Test {
  Object r1 = new Object();
  short s1 = 10; }
```

when treated as a short array a on card A_221 gives the following array contents:

```
a.length:   2         # of fields, read only
a[0]:       0x09E0    field r1, read/write
a[1]:       0x000A    field s1, read/write
```

By reading and writing the array element a[0] it was possible to directly read and write references. Similar results were obtained on the three other cards (B_22, C_211A, D_211), although in the case of C_211A we did not manage to effectively write a reference.

Pursuing this attack further we tried two more things: switching references around and spoofing references.

4.2.3 Switching References [switch] Once we have a direct access to a reference we can try to replace it (by direct assignments) with another reference, even if these have incompatible types. Our test results show that if the new value is a valid reference (i.e. existing reference) this is perfectly possible. Assume, for example, that we have these two field declarations in some class C:

```
MyClass1 c1 = new MyClass1();
MyClass2 c2 = new MyClass2();
```

Accessing an object of class C as an array, we should be able to swap the values of c1 and c2. This in turn introduces further type confusion: field c1 points to a reference of type MyClass2 and field c2 to a reference of type MyClass1.

Two cards, A_221 and B_22, allowed us to do it. It was possible to read instance fields of such switched references, but only as long as the accessed fields stayed within the original object bounds. This suggests that these cards perform dynamic bounds checks when accessing instance fields, analogous to the dynamic bounds checks when accessing array elements. We will call this *object bounds checking*. Indeed, given the similarity of memory layout for arrays

and other objects on these card, the code for accessing instance fields and array elements might be identical. Such object bounds checking prevents reference switching from giving access beyond original object bounds, and hence prevents access to another applet's data.

Another way in which reference switching might give access to another applet would be setting a field in one applet to point to an object belonging to another applet. However, here the firewall checks prevent illegal access via such a reference.

Also, methods can be called on the switched references, as long as calling such methods did not cause object bounds violations or referring to stack elements below the stack boundary.

4.2.4 AID Exploitation [AID] The possibility to switch references could be abused to manipulate system-owned AID objects. An AID (Applet IDentifier) object is effectively a byte sequence that is used to identify applets on a card. An AID object has a field which points to a byte array that stores the actual AID bytes. Changing the value of this field would change the value of an AID, whereas AIDs are supposed to be immutable.

An obstacle to this attack might be the firewall mechanism; indeed, if we try to change the field of a system-owned AID object from an applet this is prevented by the firewall. However, if we access the AID object as an array, then on cards `A_221` and `B_22` we could change the values of system-owned AIDs. This has serious consequences: a hostile applet can corrupt the global AID registry, and try to impersonate other applets. This attack is a much stronger version of the one described in e.g. [7].

Because of the privileged nature of system-owned AID objects – they are JCRE entry points – further exploits might be possible.

4.2.5 Spoofing References [spoof] Going one step further than switching references, we tried spoofing references, i.e. creating a reference from a numerical value by assigning a `short` value to a reference.

Any way we tried this, cards `A_221`, `B_22`, and `C_211A` refused to use such references: the card session was muted or an exception was thrown.

Card `D_211`, an older generation card, did let us spoof references. It was possible to write a small applet that effectively let us "read" any reference from the card by using the techniques we just described and a little bit of CAP file manipulation trickery. By "read" we mean that it is possible to get *some* value that supposedly resides at the memory address indicated by the reference. However, composing a sequence of such reads (going through all the possible reference values) did not really give a valid memory image of the card. That is, it was not possible to recognise parts of bytecode that should be on the card, or any applet data we would expect to find.

Cards can detect spoofing of references by keeping track of additional data in their representation of objects or references in memory and refusing to operate on (references to) objects if this data is not present or corrupted.

For instance, analysing our failed attempts to spoof references on `A_221` we noticed that each allocated object (even the simplest one) takes up at least 8 bytes of memory. That is, the values of references to subsequently allocated objects, when read as numerical values, differ by at least eight. An array of length 1 would even take up 16 bytes. It is clear that two bytes are used to store the number of fields (or array length, in case of an array). However, it is not clear what the other six bytes (or more in case of arrays) are used for: it will include information about the runtime type and the firewall context, but it could also contain additional checksums to check the integrity of a reference. If it would be possible to reconstruct the structure of this data (difficult because of the number of combinations to try) we believe constructing a fake reference could be considered a possibility, although an unlikely one.

We also tested references of arrays of different memory type (persistent and transient). The values of references to different kinds of arrays seem to be 'next to each other', which would indicate that the value of the reference has little to do with the actual memory address. Apparently there is an additional mechanism in the VM to map these reference to physical addresses.

4.3 More Type Confusion Attacks

Obviously, the attacks we have just described do not exhaust all possibilities. Many more are possible. For example, by changing the number of parameters in the shareable method one could try to read data off the execution stack, but this did not succeed on any of our cards.

Another example is that it is possible to reverse the type confusion between arrays and objects, and access an array as an object, with the aim to try accessing an array beyond its array bounds. Such an 'array as object' attack produced similar results as the object as array attack in Sect. 4.2.2, except that it was also possible on `A_211`, albeit harmless in the sense that it did not allow access beyond the original array bounds there.

5 Dynamic Countermeasures

Our experiments show that some VMs are much more defensive than others when it comes to executing ill-typed code. The Java Card specifications leave a lot of freedom for defensive features in the implementation of a VM. As long as the executed code is well-typed the defensive features should go undetected; the specifications are meant to guarantee that well-typed code executing on different smartcards always results in the same behaviour. However, ill-typed code is effectively out of scope as far as the Java Card specifications are concerned; when running ill-typed code as we have done, there are *no* guarantees that the same behaviour is observed on different cards, and additional defences can come to light. Below we give on overview of possible dynamic runtime checks a VM might implement.[2]

[2] The fact that Java Cards take so many clock cycles for each individual bytecode instruction [10] already suggests that Java Cards do quite a lot of runtime checks.

Runtime Type Checking Two cards from manufacturer B, cards `B_211` and `B_221`, appear to do runtime type checking, making them immune to all ill-typed code we tried. Card `A_211` also performs enough runtime type checks to make it immune to all our attacks. Still, because we were able to confuse byte and short arrays, albeit without being able to do actual harm, card `A_211` apparently does not do complete type checking at runtime.

Object Bounds Checking Any VM is required to do runtime checks on array bounds when accessing array elements. A VM could do some similar runtime checks of object bounds when accessing instance fields and invoking methods on objects that are not arrays, if it records the 'size' – the number of fields – of each object in the same way it records the length of an array. In the conversion of bytecode into CAP file format, names are replaced by numbers – `0` denotes the first field, `1` the second, etc. – which makes such a check easy to implement.

Two of the cards appear to do object bounds checking, namely `A_221` and `B_22`, as explained in Sect. 4.2.3.

Physical Bounds Checks Bounds checks on arrays (or objects) can be done using the 'logical' size of an array (i.e. its length and the array index), but can also be done using the 'physical' size of the array contents in bytes and the actual offset of an entry. For example, the contents of a `short` array `a` of length 10 takes up 20 bytes. When accessing `a[i]`, one could do a runtime check to see if $0 \leq$ `i` ≤ 10, or a runtime check to see if $0 \leq$ `2*i` ≤ 20. If the VM uses the physical offset `2*i` to look up the entry, one may opt for the latter check.

Our experiments suggests that card `A_211` performs physical bounds checks, as explained in Sect. 4.1.

Firewall Checks Firewall checks have to be done at runtime.[3] Our successful attacks on `C_211A` and `D_211` by confusing `byte` and `short` arrays in Sect. 4.1 as well as two successful AID object exploits (`A_221`, `B_22`) demonstrate that the firewall does not really provide defence-in-depth in the sense that it can compensate for the absence of a bytecode verifier.

Still, in *some* attacks the firewall is clearly a formidable second line of defence. For instance, attacks where we try to switch references are essentially harmless if the firewall prevents us from accessing object belonging to other contexts: at best an applet could then attack its own data, which is not an interesting attack. However, as the AID attack show, the firewall does not prevent all such attacks, as by accessing object as arrays we might be able to circumvent firewall checks. This shows that firewall implementations are not defensive, in the sense they do not make additional checks to catch type confusion, but then the specification of the firewall does not require them to be defensive.

It is conceivable that a defensive implementation of the firewall could prevent the attacks on `C_211A` and `D_211` described in Sect. 4.1, namely if firewall checks

[3] A system to statically check most of the firewall access rules is proposed in [2]. However, performing checks statically, at load time, is not necessarily an improvement over doing them at runtime, as our results with bytecode verification show.

	A_211	A_221	B_211	B_22	B_221	C_211A	C_211B	D_211
CAP file manipulation [§3.1]	+	+	+	+	+	−	−	+
Abusing shareable interfaces [§3.2]	+	+	+	+	+	−	−	−
Abusing transactions [§3.3]	−	−	+	−	+	+	−	+
Static protection						BCV	BCV	CL

BCV – On-card static bytecode verifier
CL – class loader disallowing use of shareable interfaces

Table 1. Loading ill-typed code

are performed on the 'physical' rather than the 'logical' level, as discussed above for array bound checks. Checking firewall rules at the 'physical level' would require that the VM can somehow tell the context for every memory cell, not just every reference. One way to do this would be to allocate a segment of memory for each security context, and then use old-fashioned segmentation as in operating systems as part of enforcement of the firewall. We found no evidence that any cards do this.

Integrity Checks in Memory Our experiments with spoofing references suggest that most cards provide effective integrity checks on references. To perform dynamic checks for array bounds, downcasting, and the firewall, the VM has to record some meta-data in the memory representation of objects, such as array lengths, runtime types and the context owning an object. Clearly, by recording and checking additional meta-data in objects (and possibly references) the VM could detect and prevent switching or spoofing of references.

6 Discussion

Table 1 summarises the result of Sect. 3, and shows which of the ways to get ill-typed code succeeded on which of the cards.

CAP file manipulation (CAP) and Shareable Interface Objects (SIO) did not succeed on `C_211A` and `C_211B`, because the on-card BCV does not allow ill-typed code or any code that uses shareable interfaces. The loader on `D_211` also refuses to load code that uses shareable interfaces.

Abusing the transaction mechanism works on cards `B_211`, `B_221`, `C_211A`, and `D_211`, which indicates that the implementation of the transaction mechanism is faulty when it comes to clearing up after aborted transactions.

Card `C_211B` was the only one on which we were unable to load any ill-typed code.

However, being able to load ill-typed code on a card does not guarantee success (from an attacker's point of view), as defensive features of the VM can still prevent the loaded code from doing any harm or executing at all. Things do *not* necessarily degenerate to the hopeless situation one has in C/C++, where you can basically do anything with malicious code.

Table 2 summarises the results of Sect. 4, listing which attacks succeeded in running and doing harm.

	A_211	A_221	B_211	B_22	B_221	C_211A	D_211
Dynamic protection	PBC	OBC	RTC	OBC	RTC		
`byte` as `short` array [§4.1]	✓	✓	–	–	–	↯	↯
Object as array [§4.2.1]	–	✓	–	✓	–	↯	↯
Reference switching [§4.2.3]	–	✓	–	✓	–	–	NA
Reference switching in AIDs [§4.2.4]	–	↯	–	↯	–	–	NA
Reference spoofing [§4.2.5]	–	–	–	–	–	–	↯
Array as object [§4.3]	✓	✓	–	✓	–	↯	↯

– impossible, ✓ possible but harmless, ↯ possible and harmful
PBC – Physical Bounds Checks, OBC – Object Bounds Checks,
RTC – Runtime Type Checking, NA – Not attempted

Table 2. Executing ill-typed code. No information is included for card C_211B, since we could not load ill-typed code on it.

Some attacks can do real damage. For cards C_211A and D_211 two different attacks are possible to access another applet's data, namely accessing a `byte` as a `short` array and accessing an array as an object. The latter attacks allows unrestricted forward memory access on C_211A. Switching references on A_221 and B_22 is possible but mostly harmless, since the firewall prevents access to data of another applet. The notable exception is using this attack to access the internals of AID objects, where the attack becomes harmful as it allows a hostile applet to alter any system-owned AIDs and redefine the entire AID registry of the card. Spoofing references on D_211 also allowed unrestricted memory access; even though the memory still seemed to be scrambled, and we could not exploit access to it in a meaningful way, we do regard this as dangerous.

One interesting conclusion is that having an on-card BCV is not all that it is cracked up to be: one of the vulnerable cards we identified has an on-card BCV. A single bug, in this case in the transaction mechanism, can completely undermine the security provided by the on-card BCV.[4] Also, cards with an on-card BCV rule out any use of Shareable Interfaces, which in retrospect is understandable, but we never realised this before we tried.

As a defensive mechanism, runtime type checking is therefore probably a more robust protection mechanisms than a static BCV. Indeed, another interesting conclusion of our work is that runtime defensive mechanisms go a long way to protect against ill-typed code, as results with card A_211, B_211, and B_221 show. The obvious disadvantage of runtime checks is the decreased performance of the JVM. However, we did not notice any considerable performance differences between our cards. More factors play a role in smartcard performance (such as the underlying hardware), so more testing would be required to establish what is the actual impact of runtime checks on performance.

We should repeat that the vulnerabilities found are of course only a problem on cards allowing the installation of additional applets. On most, if not all, Java

[4] One hopes that Sun's Technology Compatibility Kit (TCK) for Java Card includes test cases to detect this bug. Unfortunately, the TCK is not public so we cannot check that it does.

Card smartcards in the field, post-issuance download of additional applets will be disabled or at least restricted by digital signatures. Still, for security evaluations it can be extremely useful (and cost-saving) to have confidence in the fact that there are no ways for different applets to affect each other's behaviour, except when explicitly allowed by shareable interfaces.

Acknowledgements Thanks for Marc Witteman of Riscure and Olivier van Nieuwenhuyze and Joan Daemen at STMicroelectronics for their insights. We also thank Riscure for access to their extensive Java Card test suite, and STMicroelectronics for access to their CAP file manipulation tool.

The work of Wojciech Mostowski is supported by Sentinels, the Dutch research programme in computer security, which is financed by the Technology Foundation STW, the Netherlands Organisation for Scientific Research (NWO), and the Dutch Ministry of Economic Affairs.

References

1. 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.
2. Werner Dietl, Peter Müller, and Arnd Poetzsch-Heffter. A type system for checking applet isolation in Java Card. In *Construction and Analysis of Safe, Secure and Interoperable Smart devices (CASSIS 2004)*, volume 3362 of *LNCS*, pages 129–150. Springer, 2004.
3. Sudhakar Govindavajhala and Andrew W. Appel. Using memory errors to attack a virtual machine. In *IEEE Symposium on Security and Privacy*, pages 154–165, 2003.
4. Engelbert Hubbers, Wojciech Mostowski, and Erik Poll. Tearing Java Cards. In *Proceedings, e-Smart 2006, Sophia-Antipolis, France, September 20–22*, 2006.
5. Claude Marché and Nicolas Rousset. Verification of Java Card applets behavior with respect to transactions and card tears. In *Proc. Software Engineering and Formal Methods (SEFM), Pune, India*. IEEE CS Press, 2006.
6. Gary McGraw and Edward W. Felten. *Securing Java*. Wiley, 1999. Available online at `http://www.securingjava.com/`.
7. Michael Montgomery and Ksheerabdhi Krishna. Secure object sharing in Java Card. In *Proceedings of the USENIX Workshop on Smartcard Technology (Smartcard '1999), Chicago, Illinois, USA, May 10–11*, 1999.
8. Wojciech Mostowski and Erik Poll. Testing the Java Card Applet Firewall. Technical Report ICIS–R07029, Radboud University Nijmegen, December 2007. Available at `https://pms.cs.ru.nl/iris-diglib/src/icis_tech_reports.php`.
9. Sun Microsystems, Inc., `http://www.sun.com`. *Java Card 2.2.2 Runtime Environment Specification*, March 2006.
10. Dennis Vermoen. Reverse engineering of Java Card applets using power analysis. Technical report, TU Delft, 2006. Availaible at `http://ce.et.tudelft.nl/publicationfiles/1162_634_thesis_Dennis.pdf`.
11. Marc Witteman. Java Card security. *Information Security Bulletin*, 8:291–298, October 2003.