A comparison of Java Cards : state-of-affairs 2006

SENTINELS

A Comparison of Java Cards: State-of-Affairs 2006*

Wojciech Mostowski[1] Jing Pan[2] Srikanth Akkiraju[3]

Erik de Vink[2] Erik Poll[1] Jerry den Hartog[3]

Abstract

This document presents the results of a comparative study of some popular Java Cards on the market. Eight different cards from four manufacturers have been considered. The analysis has been done at two levels – (i) a documentation-based comparison, also taking other publicly available resources into account, (ii) an actual hands-on testing with software developed specifically for this purpose by the PinpasJC research team. The investigations focus on basic functionality, secure channels, the transaction mechanism, support symmetric and asymmetric cryptography, Global Platform and Open Platform compliance, and garbage and memory management.

1 Introduction

Java Card plays an increasingly predominant role in smart card projects, e.g. for identity cards and travel documents. Many vendors respond to this market expansion with dedicated products. However, by design, these products are not exactly equivalent. On top of the traditional dissimilarities such as component size, many behavioural differences can be detected both at the functional and performance level. As such, this can have impact on the portability of a solution and undermine the advantage of using Java Card.

This document presents a comparative analysis of eight commercial Java Cards available to us to date (Autumn/Winter 2006), namely `C_211A`, `C_211B`, `B_211`, `B_22`, `B_221`, `A_211`, `A_221`(two slightly different instances differing in the communication speed), and `D_22`. The evaluation took place with respect to the following criteria:

*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.

[1]Computing Science Department, Radboud University, Nijmegen, The Netherlands

[2]Department of Mathematics and Computer Science, Technische Universiteit Eindhoven, The Netherlands

[3]Department of Electrical Engineering, Mathematics and Computer Science, Twente University, Enschede, The Netherlands

Compliance to standards: Compatibility to Java Card and Global Platform

Implemented features: Communication interfaces, APDU protocols, memory management, atomicity, RMI support and on-card byte-code verification availability

Performance: Execution time of cryptographic algorithms

Limits: Transaction commit capacity and APDU buffer capacity

The investigations discussed below, give an indication of the present state of affairs regarding Java Card. For example, the choice of cards under consideration has been limited by their availability. It is also noted that some results reported in this document have been based on publicly available information only. Additionally, when it came to actual card testing, two major bottlenecks were hindering our progress for some time. First of all, getting hold of a type of Java Card in small quantities is non-trivial. Secondly, some cards are personalized with proprietary authentication keys (or, more precisely, keys that are derived following proprietary schemes). Finding out this information took a considerable amount of time (see also Section 3.1).

The rest of this document is organized as follows. Section 2 discusses the card features stated in the cards' documentations [16, 15, 2, 1, 6], regarding the four evaluation criteria above. Section 3 presents several tests (and their results) we performed with the cards to explore various features: basic card features, secure channel functionality, transaction mechanism, cryptography support and speed, RMI, garbage collection, etc. For each of the tests the methodology is briefly described. Finally, Section 4 concludes the report.

2 Card Features Based on Documentation

The Java Cards that have been considered in the research reported here are the following:

- From Manufacturer C the `C_211A` and `C_211B` cards. These cards are the only cards that we were able to buy directly from the manufacturer in small quantities. They are also the only cards that have full technical documentation that is publicly available regarding the particular Global Platform implementation. On the other hand Manufacturer C continuously sells cards known to have bugs, see comments in Section 3.8.

- From Manufacturer B the following cards: an older `B_211`, `B_22`, and `B_221`. The `B_221` card is the most advanced (supporting both Java Card 2.2.1, Global Platform 2.1.1 and contactless interface) card from Manufacturer B currently available.

- From Manufacturer A the `A_211` card and `A_221` card. For the `A_221` card we have two different instances available, the main difference being the communication speed of the contact interface. Whenever any substantial difference has been noticed between the two `A_221` cards we noted them down. Also, `A_221` is the only other card in our test set that supports the latest Java Card and Global Platform technologies.

Java Card	Company	JC API	Open/Global Platform
C_211A	Manufacturer C	2.1.1	2.0.1
C_211B	Manufacturer C	2.1.1	2.0.1
B_211	Manufacturer B	2.1.1	2.0.1
B_22	Manufacturer B	2.2	2.0.1
B_221	Manufacturer B	2.2.1	2.1.1
A_211	Manufacturer A	2.1.1	2.0.1
A_221	Manufacturer A	2.2.1	2.1.1
D_22	Manufacturer D	2.2	2.1.1

Table 1: Compliance to software standards

Java Card	EEPROM(KB)	RAM(Bytes)	ROM(KB)
C_211A	32 (30)	4096	96
C_211B	64 (—)	—	—
B_211	32 (29)	—	—
B_22	64	—	—
B_221	16/32/64	—	—
A_211	32 (30)	2300	96
A_221	72 (70)	4608	160
D_22	64	—	—

Table 2: Memory characteristics

- From Manufacturer D the D_22 card.

2.1 Compliance to Standards

Table 1 compares the cards under consideration from the point of view of specific versions of the Java Card API and Open/Global Platform standard that they support.

Table 2 provides the hardware features of the cards. The values between parentheses are the amounts of free memory available for applications once the system is loaded. The documentation from the vendor of B_211 card does not show any information about the capacity of RAM or ROM. The total amount of ROM size of the A_221 card is not presented in any documentations either,[1] though the vendor of this card's microcontroller does say that the card has 160KB of ROM.

2.2 Implemented Features

Table 3 compares the cards under consideration with respect to the data transport layer, whereas Table 4 provides an overview of the availability of additional features such as

[1]There is actually no formal documentation available for this card as other cards from Manufacturer A.

Java Card	APDU Protocols	Communication Interface
C_211A	T=0	Contact
C_211B	T=0, T=1	Contact
B_211	T=0, T=1	Contact
B_22	T=0, T=1	Contact
B_221	T=0, T=1, T=CL	Contact, Contactless
A_211	T=0, T=1	Contact
A_221	T=0, T=1, T=CL	USB 2.0 (Low Speed) Contact, Contactless
D_22	T=0, T=1, T=CL	Contact, Contactless, USB

Table 3: Communication features

Java Card	Garbage Collection	RMI Supported	On-card Byte-code Verification	Logical Channel
C_211A	—	No	Yes	No
C_211B	—	No	Yes	No
B_211	Yes	No	—	No
B_22	Yes	Yes	—	Yes
B_221	Yes	Yes	—	Yes
A_211	Full	No	—	No
A_221	Full	Yes	—	Yes
D_22	—	Yes	—	—

Table 4: Java Card and Open Platform features

garbage collection, RMI, on-card byte-code verification and logical channels. Availability of many of these cannot be decided given the information in the vendors' documentations. For example, Manufacturer B claims that their cards support run-time garbage collection. However, it remains implicit whether they concern full or partial garbage collection. Both cards from Manufacturer A supposedly provide full garbage collection (see Section 3.11 on garbage collection). Note that logical channel functionality is added only to Global Platform specification 2.1.1 as an optional feature.

2.3 Performance

As far as we are aware, there is no publication available that actually considers a performance comparison for Java Cards. An exception is [4] which, at present, treats rather outdated cards. Manufacturer A has certain documents for each of their cards (excluding A_221), where a list of performance figures can be found, though qe are not in a position to confirm these figures. Recently, a project to measure smart card performance has been initiated in France,[2] but at the moment the project is in its very early stage.

[2] http://cedric.cnam.fr/mesure/

Java Card	Transaction Buffer Size (bytes)	APDU Buffer Size (bytes)
C_211A	—	255
C_211B	—	—
B_211	—	—
B_22	—	—
B_221	—	—
A_211	512	261
A_221	—	—
D_22	—	—

Table 5: Buffer capacity limits

2.4 Limits

The limits we consider are the size of the APDU buffer and the transaction commit buffer. The APDU buffer is used to hold incoming and outgoing communication data. The transaction buffer is used to save data involved in transactions, viz. all persistent byte and short stores, as well as persistent parameters to `Util.arrayCopy`. Only a few vendors have the buffer size figures in their documentations. However, some of these figures can be retrieved directly from the card through the Java Card API, see Section 3. The figures obtained from the documentation (only a few) are listed in Table 5.

Some of the limits are not documented at all (we mention the maximum number of Java Card objects managed, maximum size of applets or load files, maximum number of load files or applets that can be installed on a card, maximum number of secure channel keys in the Issuer Security Domain).

The data we have presented so far are based solely on available documentation (if any). In the next section we describe a number of tests we performed to verify the documented data and to obtain details not included in the documentation.

3 Card Testing

For the purpose of testing a number of applications and applets have been written. We have tested some basic features (Section 3.2), Global Platform functionality, in particular secure channels (Section 3.3), the transaction mechanism (Section 3.4). Furthermore, we have established the range of cryptographic support on the cards. We also performed speed and compatibility tests (Section 3.5) and put an effort to test the basic RMI functionality of RMI enabled cards (Section 3.7), to test the GP API support (Section 3.8) and garbage collection (Section 3.11).

3.1 Preparations

First, a suitable host-side library had to be found to communicate with the cards. Our choice was to use the IBM BlueZ (RMI) Off-Card API (available as part of the IBM JCOP tool-set for Eclipse) and build our testing software on top of this library.[3] In the process of developing the software we discovered some (minor) bugs in the IBM libraries – we developed some small workarounds for these problems. Some of the programs we have written are the following:

- PATT (PinPas Applet Testing Tool) – a command line tool to manage applets on the card. The main feature of the tool is that it talks to a wide range of different Java Cards through the Global Platform interface taking into account all small 'quirks' that the cards may require (see below).
- PinPas card customiser – a very simple application that loads a couple of custom key sets into ISD to be used in the Secure Channel test. This application also performs the detailed GP key registry test, see Section 3.3.1.
- Secure Channel tester (Section 3.3) – an application that runs a set of GP commands on the card to reveal details of (Visa) Global Platform [7, 17] behaviour of the card. The results are presented to the user in concise, human readable form.
- A simple applet (Section 3.2) and host side application to retrieve basic card information from the card through the Java Card API. The host application presents the results in human readable form.
- A test applet for transactions and non-atomic methods (Section 3.4).
- An applet and host side application to test the cryptographic features of the card (Section 3.5): supported algorithms, compatibility test (with respect to the desktop Java Crypto API), and performance. Again results are presented in a human readable form.
- Global/Open Platform API test applet and host application (Section 3.8).
- Garbage collection test applet and host application (Section 3.11).

The IBM Off-Card API provides a uniform framework to manage all GP compliant cards, but has been developed by IBM and tested mainly with their own cards. Trying to make the library talk to cards from various vendors revealed some interesting card features:

- Two Manufacturer B cards (`B_211` and `B_221`) have specific initial static keys derived on a 'per card' basis from the standard GP keys (`4041...4F`). Finding out how these derivation routines work took a while. For the `B_211` card the routine (defined in

[3] Another possibility is to use an open-source Global Platform implementation – `http://sourceforge.net/projects/globalplatform/`, however, we prefer to work with Java programs and libraries.

VISA Card Personalisation Guideline we believe) can be found on the Java Card forums, although it is in principle secret/proprietary. For the B_221 card the routine is similar, but differs in small details, after a long search we found it is documented in a rather obvious place [5]. However, again, VISA itself seems to be hiding this information. During the process of figuring out these derivation schemes some cards got locked.

- The B_211 cards are somewhat more sensitive than all the other cards to the way the applets are loaded onto the card. The required loading mode is 'component by component' one.

- Both B_22 and D_22 cards implement Java Card API version 2.2 (not version 2.2.1). The Java Card SUN Development Kit version 2.2 has to be used (2.2.1 library files are not downwards compatible with 2.2 cards, although the APIs themselves are almost identical, if not actually the same) to prepare applets for these cards.

- First applet selection after card reset takes more time (noticeably long) on B_22 compared to the other cards.

- C_211A and C_211B cards do not accept standard .cap files produced by the SUN Java Card Development Kit. This is due to a 'picky' on-card byte-code verifier. The .cap files have to be transformed before loading with proprietary software (provided by Trusted Logic, it can be downloaded for free from their web site).[4]

- Rather than B_211 cards, e.g. C_211A does not accept 'component by component' loading. Instead, 'all in one' mode is required. Also, certain CAP components have to be included in the load command, a procedure that other cards do not require.

- With our reader (Omnikey CardMan 5121), for the C_211A (C_211B) card loading of large .cap files failed because of a card timeout. Our guess is that this is due to lengthy on-card byte code verification. Thus, it seems that the size of the loadable applets is limited, for this set-up, by the speed of the verifier.

The applet loading differentiators have been systematically analyzed and is summarized in Table 6. We make the following comments:

- 'Load Params', 'Install Params' – these two columns indicate whether the card requires any specific parameters to load and install commands. Notably, the C_211A and C_211B cards have to be informed about the load file size ahead of time, and also require a persistent memory usage limit to be specified during applet installation (this is marked by 'CF'). None of the other cards showed such behaviour. For these three situations can be distinguished: (a) the card does not require any applet installation parameters (–), (b) the card requires at least an empty parameter string of type 1 (c9 00) marked T1, or (c) of type 2 (c9 01 00) marked T2.

[4] http://www.trusted-logic.com/down.php

- 'Load mode' – some cards require the load components of the CAP file to start at the APDU boundary (*component by component loading* – CbC), other cards require the load APDUs to be of equal lengths, in which case the components are put one after another and the whole load block is divided at the APDU block size boundary (*all in one mode* – AiO). For most of the cards the loading mode does not matter (–).

- 'Block Size' the maximum load block size, usually limited by the APDU buffer size. Here we tested plain communication where no MACing or encoding takes place, thus the additional data that maybe required in the APDU (e.g. the MAC) is not considered.

- 'Verifier' – some cards are equipped with a byte-code verifier. In that case the CAP file needs first to be transformed with a suitable tool (for Manufacturer C cards a CAP file transformer from Trusted Logic, marked 'TL').

- 'Debug' – the same cards that are equipped with a verifier also require the debug component to be loaded onto the card. 'Yes' means the debug component is required, 'No' means the debug component cannot be included (not the case for any of the cards), '–' means that the debug component is optional.

The results in Table 6 suggest that all the cards are more or less the same with respect to applet loading. The Manufacturer C cards are one exception to this rule – this is caused by the on-card verifier. The other exception is the `B_211` card, although here we are not completely sure of the result. During the early stages of card testing the results we were getting suggested that the card requires 'component by component' loading mode. However, during later systematic tests of applet loading this requirement could not be confirmed, explaining the asterisk '*' in the table. The fact that some Manufacturer B cards require specifically derived static keys to authenticate before applet loading is not considered to be an applet loading differentiator. It is, however, an important card personalization feature.

We should add that depending on the API versions installed on the card (JC or GP), properly versioned JAR and EXP files are needed to build applets for a given card and also a matching Java Card development kit has to be used. To the best of our knowledge the JAR and EXP files are interchangeable between cards that share the same API version, which in particular means that the standard files provided by Sun or Global Platform Consortium can be used. We also did manage to use the standard Java Card development kits to prepare applets for all of the cards. All in all, apart from the CAP file transformer for the Manufacturer C cards, no proprietary software was needed to prepare applets for any of the cards.

Finally, we tested the maximal number of load files that can be loaded and instantiated on the card. The test consisted of cloning one applet 64 times and subsequently loading and instantiating it on the card. The problem with some cards turned out to be that the number of the actual load files/applets on the card is different from what the card reports with the 'get status' command. The following are the end results for all the cards:

A_221 We managed to put total of 30 load files and 29 applets on this card, but GP reports only 21 load files and all of the applets present on the card.

A_211 We managed to load all 64 test applets onto the card and to instantiate them all. The card, however, reports only 21 load files and 15 applets with the 'get status' command.

B_221 We managed to put 17 load files and 13 applets onto this card, and the card's report reflects the reality.

B_22 We managed to put 51 load files and 37 applets onto this card, and the card's report reflects the reality.

B_211 We managed to put 28 load files and 19 applets onto this card, and the card's report reflects the reality.

C_211A We managed to put 14 load files and 7 applets onto this card, and the card's report reflects the reality.

C_211B We managed to put 28 load files and 17 applets onto this card, however, the card in the end the card reported only 8 load files and 3(!) applets.

D_22 We managed to load and instantiate all 64 test applets on to the card. Moreover, the card reported all of the load files and applets in the 'get status' response.

A note about C_211A and C_211B cards is due: the number of load files and applets depend on the memory settings of an applet – the lower the limit is set for one applet the more applets can be loaded. Thus turned out to be very difficult to figure out the actual limits of those cards.

For the two cards that loaded all the test applets (A_211 and D_22) we rerun the test with 128 cloned applets. The A_211 card managed to store the total of 115 load files, the D_22 managed to load all the test files again resulting in 132 load files total on the card. From this we conclude that these two cards are simply limited by the available memory when it comes to applet loading.

Table 7 summarizes the results. Of course, because of the nature of the test those values should be treated as approximate, they rather indicate the range of the capacity of the card rather than the exact limit.

3.2 The BasicInfo Applet

The BasicInfo applet is a very simple applet that reports basic features of a Java Card accessible through the Java Card API. For the 2.1.1 cards this is limited to (a) the API version, (b) the maximum transaction commit capacity, (c) the kind of APDU protocol (T0/T1), and (d) the size of the APDU buffer. For the 2.2.* cards the applet reports the following features: (a) the API version, (b) the transaction commit capacity, (c) the

Java Card	Load param	Install param	Load mode	Block size	Verifier	Debug
A_211	–	–	–	255	–	–
A_221	–	T1	–	255	–	–
C_211A	CF	CF	AiO	255	TL	Yes
C_211B	CF	CF	AiO	255	TL	Yes
B_211	–	T2	CbC*	255	–	–
B_22	–	T2	–	255	–	–
B_221	–	T1	–	255	–	–
D_22	–	T2	–	255	–	–

Table 6: Applet loading differentiators

Java Card	Load Files	Applets	Reports
A_211	115	108	−
A_221	30	29	−
C_211A	14	7	+
C_211B	28	17	−
B_211	28	19	+
B_22	51	37	+
B_221	17	13	+
D_22	>132	>130	+

Table 7: Applet loading differentiators

available memory (persistent and transient), (d) the kind of APDU protocol (T0/T1/TCL), (e) the APDU size, and (f) the support for object deletion. Additionally, based on the card's ATR,[5] the maximum supported baud rate of the card is reported – the note 'default' means that the card does not report any particular speed, which usually means the default (initial) speed of 9600 bits/sec is used for the whole communication session. The results of running the `BasicInfo` applet on the cards are collected in Appendix A and an overview of results is given in Table 8. The 'Memory Persistent' column indicates the number of bytes of free persistent memory reported by the Java Card API call – because of the limits of the `short` data type, the Java Card API does not report memory sizes above 32K, even though some cards offer more. The 'Memory Transient' column lists the amount of free transient (RAM) memory in two modes – memory that is cleared on card reset (first value), and memory that is cleared on applet deselection (second value).[6]

We have encountered the following:

- B_221: This card reports different sizes of RAM memory for two memory clearing modes: clearing on reset and clearing on deselection. As noted later in the report

[5]The information on how to decode the supported baud rate out of the ATR is available at `http://www.cs.uct.ac.za/Research/DNA/SOCS/psec2.html`.

[6]Note, that the test was not run on cards that were totally empty. In particular, the test applet itself was present on the card, which already takes up some resources. Thus the actual memory size, especially RAM, of an empty card may differ slightly. See also Section 3.11.

(Section 3.11), there is something strange about transient memory reporting on this card.

- `D_22` and `C_211B`: These cards indicate unusual APDU buffer sizes (274 bytes and 272 bytes respectively).

For the Java Card 2.2.* API a call to `APDU.getProtocol` should in principle give the type of the protocol/mode (Wireless Type A, Wireless Type B, or USB). All the 2.2.* cards however, return the default value. According to the Java Card API documentation that means the wireless protocol is simply ISO 14443-4 (T=CL).

Finally, during multiple applet loading and deletion we obtained some strange results with respect to the card memory management and garbage collection. Apparently one of our `C_211A` cards does not clear the memory properly on applet deletion – after a sequence of applet loads and deletions the card refused to load applets, giving an error indicating lack of memory on the card. This way we managed to effectively lock one of our `C_211A` card. All other cards (including a different instance of the same `C_211A` card), despite the same loading and deleting operations, still functioned properly.

3.3 Secure Channel and (Visa) Global Platform Tests

The purpose of the Global Platform test is to reveal the details of the Global Platform implementation on the card and compliance of the card to Visa Global Platform Guidelines [17]. It is noted that only are limited number of features is tested (as described below). So, our test is *not* a comprehensive (Visa) Global Platform test suite and its results should therefore be treated with certain reserve.

In the current version, the secure channel test application performs the following tasks:

1. Based on a response to 'initialize update' command, a basic secure channel protocol is established (01 or 02).

2. Furthermore, for a given main version, all secure channel options defined in the Global Platform Specification [7] are tried out, including implicit modes. A successful exchange of 'initialize update' and 'external authenticate' (which requires a MAC) commands indicates that a corresponding explicit option is supported by the card. For the implicit modes a simple MACed communication with implicitly derived keys is initiated, but it seems that none of the cards supports implicit secure channels.

3. Once the main version and the secure channel option are established, the secure channel is subsequently tested:

    - A secure channel is initiated in all possible security levels (PLAIN, MAC, ENC+MAC, RMAC, MAC+RMAC, ENC+MAC+RMAC). For each of the security levels a basic communication with the Security Domain is performed (a communication that requires non-trivial command and response APDU).

Java Card	Baud rate	API	APDU prot.	APDU size	Commit buf.	Mem. pers.	Mem. trans.	Object del.
A_211	9622	2.1	T0	261	500	—	—	—
A_221	161290/9622	2.2	T1/T=CL	261	512	32767	1983/1983	Yes
C_211A	78125	2.1	T0	261	510	—	—	—
C_211B	156250	2.1	T0	272	512	—	—	—
B_211	53763	2.1	T1	261	505	—	—	—
B_22	161290	2.2	T1	261	896	32767	1936/1936	Yes
B_221	161290/9622	2.2	T1/T=CL	261	504	32767	614/906	Yes
D_22	161290/9622	2.2	T0/T=CL	274	511	32767	946/946	Yes

Table 8: Overview results of the BasicInfo test

- Additionally, a check if 'Begin/End R-MAC Session' commands are supported is performed. None of the cards seem to support 'Begin/End R-MAC Session' or RMACing in general.
- Finally, based on the Visa Global Platform specification, a challenge that the card should give according to the predictable challenge rules is 'guessed' and compared to the challenge returned by the card. This indicates if the card implements predictable challenges.

4. A communication over multiple logical channels is initiated. If it succeeds a test is performed to check whether a security channel can be active on multiple channels. Lack of such a possibility indicates adherence to Visa Global Platform specifications. Additionally, the maximum number of logical channels the card supports (again, based on a trial and error test) is reported.

5. An additional key set is loaded onto the card and a key deletion is attempted. This indicates the support for key deletion in the Global Platform implementation.

6. Finally, an attempt to instantiate an additional Security Domain from a standard load file (`A0000000035350`) indicated by VGP specification [17] is performed. Note, that the card may have a different load file than the one required by the VGP specification to instantiate additional Security Domains. We tested VGP compliance here only for the standard load file.

The complete test results are gathered in Appendix B. We give an overview of results in Table 9 and list some comments, where applicable:

- `C_211A`: Regarding key management on this card, the following has been observed. It seems that the card can hold multiple key sets, but only one can be active at a time – loading a new key set makes it the default one and none of the previous key sets can be used for authentication, although the card's behaviour suggests the other key sets still exist on the card. Further tests (see the next section) of different instances of the card suggest that this is caused by some sort of fault that we induced on the card during testing. Unfortunately, we are not able to reconstruct the fault that occurred.

- `D_22`: For this card, although the default load file for the security domain is present on the card, we were not able to instantiate any additional security domains – the error that we get is 'conditions of use not satisfied', which none of the other cards reported in the same scenario. So far, we did not find any information that would give us any clue as to what the reason for this behaviour could be.

One thing we find quite unexpected is that none of the cards support Visa predictable challenges. We tried two challenge calculation schemes described in two different versions of the VGP specification and none of the challenges were calculated accordingly by the cards. For the `A_221` card however the challenges *are* predictable – they can be repeated each time the associated sequence counter is reset. See the next section for details.

Java Card	GP Ver.	SCP	SCP Option	Key Deletion	VGP Chall.	Log. Channels	Additional SD
A_211	2.0.1	01	05	No	No	No	No
A_221	2.1.1	02	15	No	No	3	Yes
C_211A	2.0.1	01	05	No	No	No	No
C_211B	2.0.1	01	05	No	No	No	No
B_211	2.0.1	01	05	Yes	No	No	Yes
B_22	2.0.1	01	05	Yes	No	3	Yes
B_221	2.1.1	02	15	No	No	3	Yes
D_22	2.0.1	01	05	Yes	No	3	No

Table 9: Overview results of the Global Platform test

3.3.1 GP Key Registry Test

A separate application has been developed to perform some tests on the GP key registry. The test has four purposes:

- To see if it possible to replace only parts of key sets (say only 1 key, or only 2 keys, etc.) and to see what happens to the sequence counter associated with key set after such an operation. The sequence counter is only implemented in the Secure Channel protocol version 2, thus, the last part only applies to corresponding cards (`A_221` and `B_221`). A key is always modified/replaced with the same key.
- If the sequence counter gets reset after key replacement, it is then easy to check if the card implements predictable challenges: (A) we ask the card for a challenge, then (B) reset the counter by reloading the keys, and ask the card for a challenge again. If the two challenges match that means that the challenge is calculated based solely on the sequence counter and some other fixed data (e.g. current AID). Note again, that testing predictable challenges only makes sense for cards implementing SCP02.
- An attempt to delete only one key out of the key set is attempted. Again, if this turns out to be possible, we check (a) what happens to the sequence counter, (b) whether such 'crippled' key set is usable – an authentication is attempted.

Below we summarize the results for the different cards:

`A_221` This card exhibits very specific behaviour:

- It is possible to modify the whole key set in one go (key identifiers #1, #2, #3) without any problem.
- The sequence counter gets reset after any key replacement/modification in the given key set.
- It is possible to modify only key #2 (the MAC key) and the card still functions normally.
- It is possible to modify key #1 (AUTH+ENC) but only once. A second attempt results in an error. It is however possible to reload the whole key set again. If we try to modify key #1 and some other key (meaning a pair of keys that includes key #1) an error is always reported – a pair of keys that includes key #1 cannot be modified.
- It is possible to modify key #3 (DEK/KEK), but the whole key set is not usable after such modification – authentication cannot be performed, because the card cryptogram cannot be verified (note that the keys are always replaced with the same keys). Replacing a key pair that contains key #3 (meaning only the key pair #2,#3, because the key pair #1,#3 cannot be modified) gives the same results – authentication is not possible.

- The card *does* implement predictable challenges – the challenge is repeatable after the sequence counter gets reset. The challenge however is not calculated according to VGP specification (we tried two algorithms mentioned in different versions of the VGP specification).
- It is not possible to delete single keys in the key set (on this card key deletion is not implemented).

`B_221` This card exhibits a much more consistent behaviour:

- Any single key can be modified in the key set and the sequence counter associated with the key set gets reset. Regardless of which key is replaced, the key set functions properly all the time.
- Replacing a pair of keys (any pair) is not possible.
- Replacing the whole key set fully works and gives expected results (the sequence counter gets reset).
- The challenges are not predictable in any way – they are always different (random).
- Deletion of single keys is not possible.

`B_22` This card caused the most trouble during the test and because of that only partial results are available (in fact, it was only possible to perform the test once, and the following are the results we could extract):

- To our best recollection (we did not save the log from the test) modification of any subset of the key set (including the whole set) is possible and authentication works after this.
- Since the card implements SCP01, sequence counter and predictable challenge issues were not tested.
- Key deletion – this is where the test went wrong. The card does support the deletion of the whole key set (see Section 3.3), and, to our surprise it was also possible to delete single keys from the key set. As soon as the first key was deleted the key set became unusable (naturally), but it also affected the key registry heavily (and permanently). The default key set number 1 (which we should add was not participating in the test in any way, the whole test is performed on the key set number 2) became (was 'renamed') key set number 3F (hex). This key set can be used for authentication, but it cannot be used to load new key sets (any attempt results in 'wrong data' status word). This probably means (a wild guess), that the DEK/KEK key in key set got corrupted. Since the default key set is the only one left on the card this effectively means that no new keys can be loaded onto the card, which means no further key testing can be performed.

This behaviour clearly indicates a serious bug in key registry implementation – partial key deletion should not be possible in the first place.

`A_211` On this card only modification of the whole key set is possible – modifying one or two keys fails, and so does key deletion.

`B_211` For this card it is possible to modify the whole key set, or keys #1 and #2, all other combinations fail. Deletion of single keys is not possible (the card does support whole key set deletion).

`C_211B` The same behaviour as for the `A_211` card. We did however notice, that reloading the same key set three times leads to some corruption in the key registry – trying to use key set in question leads to a 'file invalid' error code. After this the key set is not usable, nor reloadable.

`C_211A` Again, the same behaviour as for the `A_211` card – only the whole key set can be modified and no single keys (nor whole key sets) can be deleted. This test however revealed one more interesting issue: one instance of the `C_211A` card did have problems loading new keys to the key registry as we described earlier. We could not however reproduce the same problem on the second instance (from the same order batch) of the card and the key modification test passed successfully, i.e. new key sets can be added and modified keeping old key sets usable. Interestingly enough, the card that was doing strange things with the key registry is the same card that refused to load applets at some point claiming insufficient memory. The conclusion is that: we probably did break this one instance of `C_211A` card in some mysterious way that we are not able to reconstruct (perhaps by card tearing in the transaction tests). That means that the claims we made earlier about this card (strange key registry behaviour and improper memory management) may not be necessarily true in a general case. But note also that the other Manufacturer C card (`C_211B`, previous bullet) is not free of the key registry problems.

`D_22` On this card we could modify almost all key subsets – the only key subset that is not modifiable by one command is keys #1 and #3, all the other possible subsets are modifiable and the key set is usable after this operation. Deletion of single keys is not possible, but the card does support deletion of whole key sets.

3.4 The Transactions/Non-Atomic Methods Applet

This applet performs extensive testing of the transaction mechanism and the non-atomic methods of the Java Card API. Initially the applet has been written at the SoS group of Radboud University in Nijmegen to test specific cards. The result then was that some cards exhibited highly non-deterministic behaviour, sometimes non-compliant to the Java

Card standard [9, 8]. Therefore, we have run a similar test on our cards.[7] The results are summarized below:

`A_211` This card exhibited a highly deterministic behaviour in all the tests, with the following features noticed:

- A card tear during a non-atomic method call may leave a persistent array zeroed out, which is neither the original contents of the array, nor the contents requested by the copy operation. The zero values reflect the 'unpredictable' array contents quoted in the Java Card specification.
- Non-atomic methods do by-pass/override the transaction mechanism as stipulated by the standard, but they are themselves atomic. That is, they modify all or nothing of the array elements in question. Another way of looking at it is that the card implements two different/independent transaction mechanisms. This does not violate the specification of the transaction mechanism in any way, but seems unnecessary.

`A_221` This card exhibits the same behaviour as the `A_211` card, with the additional 'twists' in the contactless mode. First, interrupting the communication with the card (card tear) always results in a 'garbage' response APDU (in contact mode there is no response APDU in such cases). We attribute this to improper behaviour of the terminal software (smart card reader driver and/or IBM JPCSC implementation). Secondly, in some test cases an explicit call to `JCSystem.abortTransaction` seems to disable further APDU communication – the card does not reply to subsequent APDUs and the card session has to be restarted.

`C_211A` This card exhibited a highly deterministic behaviour in all the tests. The transaction behaviour is compliant to the Java Card standard. It also seems to have a similar feature to `A_211` cards that zeros out arrays before copying data to them. The non-atomic methods are not atomic in the sense as `A_211` methods are.

`C_211B` The behaviour is almost the same as for the `C_211A` cards, with two exceptions: (a) we never encountered any zeros in the test array (which indicates the behaviour is *better* than `C_211A`), and (b) on occasion, when the test applet is put into an infinite loop (to do the card tear), the applet returns status word `6F00`, meaning an uncaught exception occurs inside the applet. We did not yet find the cause for that exception as the problem is non-deterministic and hard to reproduce).

[7] A note to update with respect to previous versions of this report is due here. After some private discussions with people on the Java Card Forum we were convinced that the behaviour of cards that we initially thought was incorrect is in fact intended. It was however agreed that the official Java Card specification is inaccurate and needs further updates. Thus, we no longer describe our results in terms of 'non-compliance' we simply pin-point the differences between cards with respect to the transaction mechanism. It should be stressed though that if one takes the Java Card specification literally (or 'blindly'), some cards should be considered buggy.

B_211 For this card interrupting the non-atomic method call (inside or outside of transaction) in most cases results in garbage/random data left in the array. Although this is allowed by the standard to happen when a non-atomic method is called inside a transaction, it is not allowed to happen outside of a transaction (but again,some people claim this is a 'misprint' in the specification). In all the cases when the non-atomic method is not interrupted the card seems to behave according to the Java Card standard.

B_22 This card has a very similar behaviour to B_211 when it comes to interrupting non-atomic method calls – in many cases it results with garbage/random data left in the array—sometimes the whole array is filled with FF values (hex), sometimes with random, partially modified data. In all the cases when the non-atomic method is not interrupted the card seems to behave according to the Java Card standard. It seems however, that persistent arrays are treated globally by this card, i.e. non-atomic updates to persistent arrays are atomic in the same sense as for the A_211 card, and modifying a single array element marks the whole array modified. This last part influences the card's behaviour with respect to transaction roll-back for non-atomic methods used inside transactions. Here, the card's behaviour is different from all the other cards, but is still compliant with the official Java Card specification.

B_221 This card seems to have a much more stable transaction mechanism implementation than the other Manufacturer B cards. When there are no card tears the transaction mechanism behaves deterministically and in accordance with Java Card specification. Non-atomic methods are atomic in the sense A_211 ones are. On a card tear occurring during a transaction and a non-atomic method in progress the test array can occasionally be left with zero values (not the old or the new value). A non-atomic method interrupted by a card tear usually leaves the test array in 'good shape', but occasionally a 'slightly' random data can be left in the array – by 'slightly' we mean, e.g., the whole array with an exception for one element is filled with zeros.

For this card the same problem with wireless communication has been noticed as with the A_221 card – APDU communication is not possible after an explicit transaction abort, and the card session has to be restarted.

D_22 Here, the beginning of the test went fine and for a while indicated that the card behaves correctly and according to the specifications. The card did not even exhibit the problem we had with B_221 and A_221 in contactless mode (unresponsiveness after an explicit abort). However, in the middle of the test one of the card tears caused the test array to be zeroed out (behaviour noticed with other cards before). For this card however, the card manager decided to lock the test applet after the tear. We reinstalled and rerun the test and ended up with the exact same situation – our test applet was locked by the card manager again. On the third attempt something more went wrong – the card started returning an unusual status word[8] on

[8]In fact, what we think that happens is that the card sends back a legitimate APDU with a correct

ISD selection and ISD authentication was no longer possible (the card still returns an answer to 'initialize update' but on 'external authenticate' it returns an error status word declining authentication). In effect the card became useless – neither ISD nor any other previously installed applets are selectable. As far as we see it – the card has a deeply 'inadequate' transaction mechanism implementation. We should stress here that the problems occurred quite quickly only after a dozen or so card tears, while all the other cards managed to survive over 100 tears each. Because of this results we are not going to attempt any more transactions tests on this type of card.

3.5 The CryptoTest Applet

This applet performs extensive cryptographic testing of the Java Card API, including key generation, data encryption/decryption, MAC signature generation/verification and hash generation using message digest algorithms. For the convenience, all the input data (e.g. plaintext) used in our tests are in fact 128 bytes long arrays. However, extending the applet for larger data is not a difficult task.

The time for processing the test APDUs has been measured. The overhead time for APDU transmission and processing was also measured and has been subtracted in the test results. The overhead time here is seen as the time spent on a full execution of the application – APDU transmission and processing, output computation and APDU returning, minus all the above except for a call of the method that actually does the cryptography operation, e.g., an API call of `Cipher.doFinal()` or `Signature.sign()`, etc. Every test application has been performed 10 times in a loop in order to get an average value. Due to the 'fresh' allocation of necessary objects and arrays on the first time when an APDU is processed, in fact every card always spends much more time on the first processing than on the subsequent ones. In order to get rid of this overhead, each APDU is performed 11 times and the results from the first time is deleted. The indicated time in our test results is for one processing of a cryptographic operation. We notice that some of the results may not make much sense, e.g., triple DES takes less time than single DES. However, we believe we have reached the boundaries on overhead measurement on software level. If a more accurate result is required, power measurement equipment may be needed. Not all of the combinations of cards and algorithms could be tested, since the given algorithms were not implemented on every card.

3.5.1 What has been tested

The packages `javacard.security` and `javacardx.crypto` define a set of classes and interfaces for the Java Card security and cryptography framework. All the symmetric algorithms and related keys specified in the Java Card API [11, 12, 13] are listed below and those marked with * are required by Visa GlobalPlatform [17]. Any possible combination of them have been applied in our tests.

status word, but with some garbage data appended to the APDU. The last two bytes of this garbage data happen to be `6283`, which can be interpreted as an unusual status word.

Cipher Algorithms in the `Cipher` Class

ALG_AES_BLOCK_128_CBC_NOPAD	ALG_AES_BLOCK_128_ECB_NOPAD
ALG_DES_CBC_ISO9797_M1*	ALG_DES_ECB_ISO9797_M1*
ALG_DES_CBC_ISO9797_M2*	ALG_DES_ECB_ISO9797_M2*
ALG_DES_CBC_NOPAD*	ALG_DES_ECB_NOPAD*
ALG_DES_CBC_PKCS5	ALG_DES_ECB_PKCS5
MODE_DECRYPT*	MODE_ENCRYPT*

Interfaces for Symmetric Keys

AESKey	DESKey*	Key*	SecretKey*

Keys in the `KeyBuilder` Class

LENGTH_AES_128	LENGTH_DES*
LENGTH_AES_192	LENGTH_DES3_2KEY*
LENGTH_AES_256	LENGTH_DES3_3KEY
TYPE_AES	TYPE_DES*
TYPE_AES_TRANSIENT_DESELECT	TYPE_DES_TRANSIENT_DESELECT*
TYPE_AES_TRANSIENT_RESET	TYPE_DES_TRANSIENT_RESET*

MAC Algorithms in the `Signature` Class

ALG_AES_MAC_128_NOPAD	
ALG_DES_MAC4_ISO9797_1_M2_ALG3	ALG_DES_MAC8_ISO9797_1_M2_ALG3
ALG_DES_MAC4_ISO9797_M1	ALG_DES_MAC8_ISO9797_M1*
ALG_DES_MAC4_ISO9797_M2	ALG_DES_MAC8_ISO9797_M2*
ALG_DES_MAC4_NOPAD	ALG_DES_MAC8_NOPAD*
ALG_DES_MAC4_PKCS5	ALG_DES_MAC8_PKCS5
MODE_SIGN*	MODE_VERIFY*

Hashing Algorithms in the `MessageDigest` Class

ALG_MD5	ALG_RIPEMD160	ALG_SHA*

3.5.2 Symmetric Encryption and Decryption

These tests involve applications of cipher algorithms with keys shown in Section 3.5.1.

Data Encryption The host initializes the conversation by passing the plaintext to the applet and assigning the algorithm to be used in this application. The applet first checks whether the desired algorithm is available on the card. If it is not, an agreed error code will be returned to the host. Otherwise, the applet generates a secret key and encrypts the plaintext using the required algorithm and the key. Finally, it returns the ciphertext, the key and any parameters needed for decryption. The host

then decrypts the ciphertext from the applet and compares the decrypted data with the original plaintext it passed to the applet. If they match the card is considered to hold a correct implementation of data encryption for the specific algorithm; the implementation is seen to be incorrect otherwise.

Data Decryption The host generates a secret key this time, encrypts the plaintext and sends to the applet the ciphertext, the key and any other parameters as needed. The applet checks the availability of the desired algorithm and returns an error code in case it is not supported. Otherwise, it decrypts the ciphertext using the knowledge from the host and returns the decrypted data to the host. The host then compares the data with the original plaintext. If they match the corresponding implementation in card is considered to be correct, and it is not otherwise.

On the host side, implementations for all the cipher algorithms can be found in the SunJCE in JDK 5.0 [14], except the ones for ISO 9797 Padding Method 1 and padding Method 2 [10] that are missing. We employ the implementation from Bouncy Castle [3] for padding Method 2 [10]. The former one we implemented ourselves, since no existing implementation was found. The results are listed in Table 10, 11, 12 and 13. In these tables time is in milliseconds and is indicated as data encryption or decryption. Categories marked with * are required to be implemented for Visa compliance.

`A_211` This card works the slowest among the tested cards. Together with `C_211A` and `C_211B`, it also has the least number of implementations of algorithms. The speed for encryption and decryption are also quite unbalanced, i.e. there is a big gap between the values.

`A_221` This card turns out very slow after the overhead time is filtered out, and apparently the encryption for AES is much more time-consuming than decryption.

`C_211A` As an JC21 card (card that supports Java Card Platform 2.1), it shows a quite satisfying results w.r.t. speed.

`C_211B` As a sibling of `C_211A`, it supports the same algorithms as `C_211A` does. This card, however, is obviously slower than the other one. Some results on triple DES with two keys also show discrepancy between the time for encryption and the time for decryption.

`B_211` Encryption in this card is significantly slower than decryption on the corresponding algorithms. Like the other Manufacturer B cards, triple DES with double keys consumes approximately the same amount of time as single DES, while triple DES with three keys requires much more time. The good thing about this card is that it has the implementation of all the cipher algorithms specified in the Java Card API 2.1.1 [11].

Card	DES (Single DES)							
	No Padding		PKCS#5 Pad.		ISO9797 M1 Pad.		ISO9797 M2 Pad.	
	CBC*	ECB*	CBC	ECB	CBC*	ECB*	CBC*	ECB*
A_211	335/405	326/328	—	—	335/336	326/396	430/480	422/337
A_221	355/353	355/354	—	—	354/354	355/354	375/361	376/361
C_211A	41/41	37/37	—	—	40/42	37/38	45/44	40/39
C_211B	71/73	70/70	—	—	71/71	71/87	82/73	76/68
B_211	188/157	175/136	233/164	217/140	188/156	176/134	233/164	218/140
B_22	67/69	65/67	71/69	68/67	67/68	67/67	71/69	68/67
B_221	46/46	44/43	47/46	47/46	46/49	44/44	48/47	45/46
D_22	41/40	11/13	—	—	41/40	12/14	40/42	12/11

Table 10: CryptoTest result for DES

Card	3-DES (Triple DES with 2-Keys)					
	No Padding		PKCS#5 Pad.		ISO9797 M1 Pad.	ISO9797 M2 Pad.
	CBC*	ECB*	CBC	ECB	CBC*	CBC*
A_211	408/407	332/399	—	—	408/407	367/483
A_221	356/355	356/355	—	—	356/356	378/363
C_211A	58/60	55/55	—	—	58/60	62/62
C_211B	73/73	71/72	—	—	73/91	81/73
B_211	189/160	177/136	233/164	220/144	189/160	233/164
B_22	69/70	67/69	73/71	74/69	72/70	74/67
B_221	46/48	45/48	49/51	49/46	48/47	49/49
D_22	40/41	12/13	—	—	40/42	43/44

Table 11: CryptoTest result for 2-Key 3-DES

Card	3-DES (Triple DES with 3-Keys)					
	No Padding		PKCS#5 Pad.		ISO9797 M1 Pad.	ISO9797 M2 Pad.
	CBC	ECB	CBC	ECB	CBC	CBC
A_211	—	—	—	—	—	—
A_221	357/368	357/356	—	—	360/357	379/365
C_211A	—	—	—	—	—	—
C_211B	—	—	—	—	—	—
B_211	268/233	252/216	316/248	304/224	269/233	316/248
B_22	164/167	167/163	176/173	174/172	166/167	178/172
B_221	58/62	62/60	65/63	64/62	61/63	63/64
D_22	43/40	13/13	—	—	41/41	44/42

Table 12: CryptoTest result for 3-Key 3-DES

Card	AES (128 bits Key)		AES (192 bits Key)		AES (256 bits Key)	
	No Padding		No Padding		No Padding	
	CBC	ECB	CBC	ECB	CBC	ECB
A_211	—	—	—	—	—	—
A_221	243/199	239/199	239/201	244/199	242/203	241/201
C_211A	—	—	—	—	—	—
C_211B	—	—	—	—	—	—
B_211	—	—	—	—	—	—
B_22	77/89	80/90	80/96	83/95	85/100	88/102
B_221	—	—	—	—	—	—
D_22	—	—	—	—	—	—

Table 13: CryptoTest result for AES

B_22 This is the only card that supports all the algorithms in our tests. Like A_221, it also gives poor balanced results on AES operation. However, the situation here is in the other way around – the decryption consumes more time than the encryption.

B_221 This card behaves very stable in the sense that there is nothing odd to mention here. It also shows a good speed result, although it is pity that implementation of AES is missing.

D_22 The card does not show much difference between the results for DES and triple DES. However, the gap between the speed for CBC mode and for ECB mode is relatively large.

During the tests, we discovered that some properties in the Java Card Platform are remained open (i.e. unspecified) or unclear in the specification and the cards behave variously on them.

- On the Java Card platform encryption/decryption output is generated by invoking the method:

  ```
  doFinal(byte[] inBuff, short inOffset,
          short inLength, byte[] outBuff, short outOffset);
  ```

  in the Cipher class. It is, however, not specified what should happen if the value of inLength (the length of the data to be encrypted/decrypted) is zero. In fact, not all of the cards behave the same in this situation – A_211 allows zero length and while no output is generated; C_211A, C_211B, B_22, B_221, and A_221 throw a CryptoException.

- The notes on the method Cipher.doFinal() in the Java Card API says that:

  > On decryption operations (except when ISO 9797 method 1 padding is used), the padding bytes are not written to outBuff.

  What we would like to know is what happens when ISO 9797 method 1 padding is used. And, in fact, the cards are divided into two groups according to this – in A_211 and A_221 padding bytes are removed from the output while in C_211A, C_211B, B_22, B_221 and D_22 the padding bytes are kept.

- Another issue is about the setIncomingAndReceive() method in the APDU class. In the Java Card API it says:

  > APDU buffer[5..] is undefined and should not be read or written prior to invoking the setIncomingAndReceive() method if incoming data is expected. Altering the APDU buffer[5..] could corrupt incoming data.

We found that this is the case expect for `A_221` and `B_221`. In these two cards, the APDU buffer[5..] hold the exact incoming data regardless whether the method `setIncomingAndReceive()` has been invoked, and they can work as if the method has been called. Missing `setIncomingAndReceive()` on some other cards (even if there is no input data to be read in), however, leads to very strange things, for example:

- on `C_211A` cards an API method invoked in the applet's `process` method can be called twice (in some very specific situations) – exactly as if it were repeated in the source code twice.
- on `C_211A` cards and `A_211` card, when calling `setOutgoingAndSend()`, the host-side JPCSC library reports low-level errors.

What we are trying to say is that it is not only the APDU data that gets corrupted if the call to `setIncomingAndReceive()` is missing, but some other, possibly serious, card specific artifacts can occur as well.

3.5.3 Symmetric MAC Generation and Verification

On symmetric MAC generation (except when ISO 9797 MAC algorithm 3 [10] is used), the n-byte signature of a message is identical to the most significant n bytes of the last block of the encrypted data out of the message using the corresponding algorithm. For example, the signature generated by algorithm `Signature.ALG_DES_MAC_8_NOPAD` is the same as the last block of the encrypted data generated by algorithm `Cipher.ALG_DES_CBC_NOPAD`. The following applications have been applied to all possible combinations of MAC algorithms and keys shown in Section 3.5.1.

MAC Generation The application initializes from the host by asking the card to generate the MAC of given data using a specific algorithm. The applet then creates a key, generates the MAC and returns the key and the MAC back to the host. The host generates a MAC itself and compares it with the one from the applet. If the matching succeeds the implementation is seen to be correct and vice versa. The applet returns an error code if the algorithm is not implemented on the card.

MAC Verification The host generates a key and yields a MAC out of the input data using a specific algorithm. It then passes to applet the input, the signature and other necessary parameters. The applet verifies the MAC using a method call on the Java Card Platform and returns the result (yes or no) back to the host. Like the previous application, a known error code will be returned if the desired algorithm is not available.

Apart from ISO 9797 padding Method 1 and Method 2, the SunJCE [14] does not provide implementations for ISO 9797 MAC algorithm 3 [10] either. We therefore employ the implementation from Bouncy Castle [3] for the related applications. The test results

Card	DES (Single DES)							
	No Padding		PKCS#5 Pad.		ISO9797 M1 Pad.		ISO9797 M2 Pad.	
	MAC4	MAC8*	MAC4	MAC8	MAC4	MAC8*	MAC4	MAC8*
A_211	—	14/29	—	—	—	14/29	—	18/35
A_221	—	13/29	—	—	—	14/29	—	19/35
C_211A	—	50/49	—	—	—	50/50	—	53/51
C_211B	8/15	9/17	—	—	12/13	13/17	12/14	12/13
B_211	182/183	180/185	223/226	224/226	180/185	182/184	198/226	224/228
B_22	67/80	68/80	70/83	71/83	67/79	67/80	120/134	71/83
B_221	25/26	27/25	27/24	27/26	27/26	25/25	27/27	27/26
D_22	35/35	35/36	—	—	35/35	35/37	35/35	35/32

Table 14: CryptoTest result for DES MAC

Card	3-DES (Triple DES with 2-Keys)							
	No Padding		PKCS#5 Pad.		ISO9797 M1 Pad.		ISO9797 M2 Pad.	
	MAC4	MAC8*	MAC4	MAC8	MAC4	MAC8*	MAC4	MAC8*
A_211	—	15/30	—	—	—	15/30	—	19/34
A_221	—	14/30	—	—	—	14/30	—	20/35
C_211A	—	69/67	—	—	—	67/67	—	71/69
C_211B	14/16	14/19	—	—	13/15	15/18	16/16	30/23
B_211	183/186	183/187	226/228	225/229	181/186	182/187	227/228	225/228
B_22	69/82	70/82	72/87	73/85	68/86	69/82	122/135	73/85
B_221	28/27	28/27	29/27	27/28	28/27	28/27	29/27	28/28
D_22	32/34	33/32	—	—	31/33	33/34	31/33	32/34

Table 15: CryptoTest result for 2-Key 3-DES MAC

are shown in Table 14, 15, 16, 18 and 17. The time is indicated as signature verification/generation in milliseconds. Categories marked with * are required to be implemented for Visa compliance.

A_211 Together with C_211A, this card supports the least algorithms among all the cards. However, the supported algorithms show rather good results regarding the speed. What appears hard to understand here is that the time for MAC verification is twice as much as it for MAC generation, as it is expected that signature verification equals the signature generation plus short byte array comparison. Apparently this card does something extra in verification.

A_221 This card does not support any 4-byte MAC algorithms and ISO 9797 MAC algorithm 3. Like A_211, it gives quite distinguishable results for MAC verification and generation. But it is one of the fastest cards among all the cards in this scenario.

C_211A As said above, this card has the least implementations of MAC algorithms. Unlike A_211, its speed in signature operation appears not look that good.

Card	3-DES (Triple DES with 3-Keys)							
	No Padding		PKCS#5 Pad.		ISO9797 M1 Pad.		ISO9797 M2 Pad.	
	MAC4	MAC8*	MAC4	MAC8	MAC4	MAC8*	MAC4	MAC8*
A_211	—	—	—	—	—	—	—	—
A_221	—	14/30	—	—	—	14/30	—	20/35
C_211A	—	—	—	—	—	—	—	—
C_211B	—	—	—	—	—	—	—	—
B_211	260/265	261/266	307/312	308/309	260/264	261/266	308/311	308/310
B_22	165/178	165/178	170/187	174/187	164/177	164/177	222/235	174/188
B_221	41/40	42/40	43/39	41/42	42/40	40/41	43/42	42/43
D_22	33/34	35/36	—	—	36/32	36/34	35/31	35/34

Table 16: CryptoTest result for 3-Key 3-DES MAC

Card	ISO9797 ALG3 M2 Pad.	
	3-DES (Triple DES with 2-Keys)	
	MAC4	MAC8
A_211	—	—
A_221	—	—
C_211A	—	—
C_211B	—	—
B_211	—	—
B_22	119/120	120/118
B_221	28/28	28/29
D_22	4/4	4/7

Table 17: CryptoTest result for ISO 9797 MAC Algorithm 3 with 2-Key 3-DES

Card	AES (128 bits Key)	AES (192 bits Key)	AES (256 bits Key)
	No Padding	No Padding	No Padding
	MAC16	MAC16	MAC16
A_211	—	—	—
A_221	14/36	14/35	14/36
C_211A	—	—	—
C_211B	—	—	—
B_211	—	—	—
B_22	77/96	79/98	82/101
B_221	—	—	—
D_22	—	—	—

Table 18: CryptoTest result for AES MAC

`C_211B` This card seems having very quick responses to all the MAC algorithms it supports. Unlike in the tests for cipher algorithms, it is much more efficient this time than its sibling `C_211A`.

`B_211` This card seems to be significantly slower than all the rest. However, it owns the implementation of all the algorithms it could support (see [11]). Like the other two Manufacturer B cards, operations on triple DES with two keys are almost as fast as single DES but triple DES with three keys appears to be very slow compared with them.

`B_22` This card supports all the algorithms described in Java Card API 2.2 [12], although it is the slowest in all the JC22 cards. What also looks odd is that processing 4-byte MAC operations with ISO 9797 padding method 2 consumes much longer time than the corresponding 8-byte ones. We do not have an explanation for this at this moment.

`B_221` This card performs more 'reasonably' and stable than all the rest. It also gives good speeds according to what the results show. However, the lack of implementation of AES is still a blemish in an otherwise perfect thing.

`D_22` Like in the cipher algorithm tests, results here also show that this card use approximately the same amount of time for single DES and Triple DES. It is also shown that the computation on the ISO 9797 MAC algorithm on this card is *unbelievably* fast.

We also found something during our test worthy to bring up for discussion.

- In Java Card Platform, a MAC signature is generated by invoking the method

  ```
  sign (byte[] inBuff, short inOffset,
        short inLength, byte[] sigBuff, short sigOffset);
  ```

  in the `Signature` Class, and a signature is verified via method:

  ```
  verify(byte[] inBuff, short inOffset, short inLength,
         byte[] sigBuff, short sigOffset, short sigLength);
  ```

  Like what happened in the method `doFinal()`, when `inLength` is zero `A_211` does not complain and no output of MAC is generated, while all the other cards do not allow this.

- The application of the ISO 9797 MAC algorithm 3 requires the following six steps: padding, splitting the padded data into blocks, initial transformation where simple DES is applied on the first block, iteration where DES in CBC mode is applied on the second block and so on, output transformation and truncation. The combination of the initial transformation and the iteration can be seen as an overall DES application in CBC mode with zero bytes initialization vector (IV). Two methods exist in the Java Card Platform for signature initialization:

Card	MD5	RIPE MD-160	SHA*
A_211	46	—	50
A_221	26	—	32
C_211A	75	—	139
C_211B	—	—	88
B_211	56	112	71
B_22	116	131	119
B_221	33	75	51
D_22	16	—	19

Table 19: CryptoTest result for Message Digest

```
init(Key theKey, byte theMode,
     byte[] bArray, short bOff, short bLen);
```

where parameter value for the IV can be customized, and

```
init(Key theKey, byte theMode);
```

when no initialization parameters are needed or the default IV value (zero bytes) will be used for algorithms in CBC mode. To initialize a signature for ISO 9797 MAC algorithm 3, the latter method should be applied, as well as the former one with the resulting IV value being zeros. However initialization with the former method with an arbitrary value of IV passed in is in fact allowed for this algorithm. Signature can also be generated with this initialization without any complaints from the JCRE. This signature, of course, is not correct but can be mistaken as a correct one. We therefore suggest that necessary checking for consistency of algorithms and initialization parameters should be introduced here into the former method to prevent such illegal initializations of a signature.

3.5.4 Message Digest Tests

The application for message digest test is substantially simpler compared with the previous two. Here, no key is involved and only one hash code generation method in the `MessageDigest` class in the Java Card Platform exists for testing. The host generates the hash code out of the input data using one of the hashing algorithms in Section 3.5.1. The host then passes the input data and the hash to the applet. The applet hashes the input data using the specific algorithm and compares the result with the hash the host generated. If they match, the implementation in the card is seen to be correct, and otherwise not.

The results are shown in Table 19. The `D_22` card is significantly faster on this than the rest. Speaking of the speed, we notice that a card that works faster on one algorithm is not necessarily faster for another algorithm as well (e.g. `C_211A` vs. `B_22`).

3.6 Asymmetric Crypto Test

In this section we describe the results of a comprehensive asymmetric cryptographic testing. We investigated the following:

- Whether all features specified by the Java Card API are supported.
- Whether the implementation is the correct one/can be verified on the host side.

The length of the input data for the ciphers used in the test was derived based on the size of the key and the padding scheme used. For the `NoPadding` scheme, data length is as long as the key. For PKCS1 padding, the padding data needs to be a minimum of 11 bytes (88 bits) long, so the data length taken was `keyLength-11` bytes. For the rest of the padding schemes the data length taken was arbitrary, in our case 15 bytes (120 bits) for testing purposes.

Wherever possible timings were recorded. The overhead times were measured and calculated separately for APDU transmission and processing and subsequently subtracted from the total times for the actual timings of the cryptographic operations (method call like `doFinal()` for encryption and decryption, `sign()` for signature and `verify()` for verification). To get a better estimate of the timings, the cryptographic operation was performed several times in a loop and the average of the timings from the different iterations was calculated and documented. Every card spent quite some time for the first processing. So the operations were performed $n + 1$ times (n being the desired number of iterations) and the results from the first iteration were discarded for a fair estimate.

Not all of the combinations of algorithms and padding scheme that are theoretically possible, could be tested as some of them were not implemented either on the card or on the host side. If the algorithms are not present on the card then 'function not supported error code' was returned and documented. If the algorithm could not be implemented on the host side for verification then the card was made to process the data and return true if function was supported or false if it was not supported, i.e. on-card testing only was performed.

3.6.1 Tested Modules

To test the ability of the card extensively, an applet and a Java application were developed. The first aim of the test is to review the following four points:

- the ability to (a) build keys of different key lengths and (b) use different asymmetric algorithms on the card,
- the ability to generate keys of different key lengths,
- the ability to initialize cipher with different padding schemes,
- if the card supports all cryptographic operations specified by the Java Card API and, in particular, Visa Global Platform.

Apart from testing the presence of a function we also analyzed the correctness of a given function and reported the time taken to perform the actual operation (excluding the overhead times). Here we cross check the implementation on the card with that in Java development kit. Other than this, it is also scrutinized to see if all the specifications mentioned in the Java Card APIs are followed. Performance tests are repeated several times for accuracy in timings. Comparison is drawn between various cards in terms of the supported features and the timings.

The tested modules are given below. All the asymmetric algorithms, modes, key types, key lengths and signature algorithms from different classes have been mentioned below. We applied any possible combinations of them in our tests. Not all of them are required for VISA compliance, * represents those that are.

Cipher Mode in the `Cipher` Class

MODE_ENCRYPT*	MODE_DECRYPT*	

Cipher Algorithms in the `Cipher` Class

ALG_RSA_NOPAD*	ALG_RSA_PKCS1_NOPAD*	ALG_RSA_ISO97976
	ALG_RSA_PKCS1_OAEP	ALG_RSA_ISO14888

Keys in the `KeyBuilder` Class

LENGTH_RSA_512*	LENGTH_RSA_1536	LENGTH_DSA_512
LENGTH_RSA_768*	LENGTH_RSA_1792	LENGTH_DSA_768
LENGTH_RSA_1024*	LENGTH_RSA_2048	LENGTH_DSA_1024
LENGTH_RSA_1280		

Type of keys in the `KeyBuilder` Class

TYPE_RSA_PRIVATE*	TYPE_DSA_PRIVATE	TYPE_RSA_CRT_PRIVATE*
TYPE_RSA_PUBLIC*	TYPE_DSA_PUBLIC	

Mode in the `Signature` Class

MODE_SIGN*	MODE_VERIFY*

Signature Algorithms in the `Signature` Class

ALG_RSA_MD5_PKCS1	ALG_RSA_SHA_ISO9796*	ALG_RSA_RIPEMD160_ISO9796
ALG_RSA_MD5_RFC2409	ALG_RSA_SHA_PKCS1*	ALG_RSA_RIPEMD160_PKCS1
ALG_RSA_MD5_PKCS1_PSS	ALG_RSA_SHA_RFC2409	ALG_RSA_RIPEMD160_PKCS1_PSS
	ALG_RSA_SHA_PKCS1_PSS	
	ALG_DSA_SHA	

3.6.2 Asymmetric Encryption/Decryption

In this subsection the results from the testing of on card encryption and decryption are tabulated and presented.

Data Encryption The host initializes the conversation by (a) generating a pair of keys and sending and initalizing them on the applet, (b) passing the plaintext to the applet and assigning the algorithm expected to be used in the test. The applet first checks whether the desired algorithm is available on the card. If it is not, an agreed error code is returned to the host. Otherwise, the applet encrypts the plaintext using the required algorithm with the public key and returns the ciphertext back to the host. The host then decrypts the ciphertext from the applet and compares the decrypted data with the original plaintext it passed to the applet. If they match, the card is considered to hold a correct implementation of data encryption for the specific algorithm else the implementation is seen to be incorrect.

Data Decryption The situation is very similar to encryption, here the server sends the ciphertext to the applet, the applet decrypts it with the private key and sends the plaintext back to the host. The host then compares the result from applet with its own copy of the plaintext.

On the host side the algorithm implementations can be found in SunJCE in JDK 5.0 [14]. Some of the missing algorithms were found in Bouncy Castle Provider [3]. Few other algorithms were self implemented by us like signature algorithms with ISO9796 padding. It was noticed that for the encryption operation other than for `NoPadding` scheme, the verification of the ciphertext failed on the host.[9]

The results are presented in the tables 20 to 26. The time is indicated as Encryption/Decryption in milliseconds. Categories marked with * are required to be implemented for Visa compliance. N means RSA, C means RSA CRT, NV means Not Verified, — means Not Supported, CR means Crashed.

For the testing of padding scheme ISO14888, an APDU was sent to test it's presence on the card. For all the cards it returned a function not supported error code. It was concluded that none of the cards supported the particular padding scheme.

Another padding scheme ISO9796, was found not to be implemented in any of the cards for encryption/decryption algorithms. All of them returned with a function not supported error. Hence, the scheme was not included in the table.

The `B_221` card reported low level errors (the reader driver reported transmission errors) and crashed when dealing with key length 2048 bits. So it was concluded that it supports the particular key length but it behaves strangely when using it in practice. This was the only card to show such behavior.

[9] In fact, for certain algorthims all of the cards return easily spottable wrong results – a repeated encryption of the *same* plaintext with the *same* keys gives always different ciphertexts in subsequent runs. To this date, we are not able to explain this behaviour. Neither we are able to explain the other reported failed results in the reminder of this section, even though they seem to follow certain patterns.

It is also noticeable that none of the cards implement all of the algorithms for all key lengths. The closest is the D_22 card which implements three of the five padding schemes for all key lengths. The C_211A card and the B_211 card support the least number of schemes. They do not support key lengths greater than 1024 bits. C_211B card supports only 1024 bits and 2048 bits key lengths and nothing in between.

A_211 This is one of the slowest cards along with B_22 and C_211A. It is slowest in both encryption and decryption. Though it has the most implementations among all the older specification cards. Like all cards, encryption in asymmetric algorithms is faster than decryption. This is very apparent as the length of the key increases. Also decryption in CRT mode is significantly faster than NON-CRT mode. Encryption takes about same time in both modes.

A_221 This is one of the fastest cards along with B_211 among all the cards tested, as far as encryption is concerned. It is almost twice as fast as the next fastest card. Decryption though, takes almost the same time on most cards. The trend, RSA-CRT mode being faster in decryption mode and same time in both modes for encryption, continues.

C_211A This along with C_211B card has the least implementations. This card does not support any key length greater than 1024 bits. It is also one of the slowest cards.

C_211B This card has the least number of implementations. It supports only key lengths 1024 and 2048. It is also the slowest card. For key length 2048 bits, this card is almost twice as slow as the next slowest card.

B_211 This is the fastest card for encryption. But it does not support key lengths greater than 1024. Decryption behaves as expected and the timings are comparable to other cards.

B_22 This is one of the slower cards. Apart from that it shows no odd behavior.

B_221 This card showed strange behavior when dealing with key length of 2048 bits. The applet kept crashing when computing the encryption/decryption for more than once.

D_22 This card supports the most implementations. It is also good in terms of speed of the cryptographic operation.

3.6.3 Building Keys

Here the card was tested with different key lengths to see what key lengths are supported by the buildKey method. It is also to be noted that if the building failed then we bailed the card out for all remaining tests for that particular key length. This is due to the fact that verification of the implementation of encryption, decryption, signature, verification all require building of the key to be supported for that particular length.

Card	RSA (KeyLength 512*)					
	No Padding		PKCS#1 Pad		OAEP Pad	
	N*	C*	N*	C*	N	C
A_211	35/90	33/62	NV/87	NV/58	–	–
A_221	8/48	9/33	NV/47	NV/32	–	–
C_211A	42/94	37/101	NV/94	NV/100	–	–
C_211B	–	–	–	–	–	–
B_211	18/197	18/207	NV/199	NV/206	–	–
B_22	54/137	53/160	NV/136	NV/161	–	–
B_221	59/143	58/107	NV/149	NV/111	–	–
D_22	15/70	16/48	NV/71	NV/47	NV/81	NV/55

Table 20: RSA crypto key-length 512 bit

Card	RSA (KeyLength 768*)					
	No Padding		PKCS#1 Pad		OAEP Pad	
	N*	C*	N*	C*	N	C
A_211	44/220	43/106	NV/211	NV/101	–	–
A_221	12/129	12/61	NV/129	NV/64	–	–
C_211A	46/212	43/157	NV/213	NV/157	–	–
C_211B	–	–	–	–	–	–
B_211	28/266	27/207	NV/263	NV/205	–	–
B_22	55/266	57/219	NV/268	NV/219	–	–
B_221	59/291	59/161	NV/294	NV/162	–	–
D_22	24/188	24/89	NV/186	NV/88	NV/204	NV/103

Table 21: RSA crypto key-length 768 bit

Card	RSA (KeyLength 1024*)					
	No Padding		PKCS#1 Pad		OAEP Pad	
	N*	C*	N*	C*	N	C
A_211	52/443	53/177	NV/441	NV/175	–	–
A_221	11/279	11/110	NV/276	NV/110	–	–
C_211A	54/445	54/245	NV/441	NV/243	–	–
C_211B	88/247	84/239	NV/242	NV/236	–	–
B_211	25/365	24/213	NV/364	NV/215	–	–
B_22	63/502	60/322	NV/500	NV/319	–	–
B_221	59/560	59/245	NV/561	NV/248	–	–
D_22	28/399	32/162	NV/399	NV/156	NV/420	NV/181

Table 22: RSA crypto key-length 1024 bit

Card	RSA (KeyLength 1280)					
	No Padding		PKCS#1 Pad		OAEP Pad	
	N*	C*	N*	C*	N	C
A_211	62/803	63/277	NV/798	NV/272	–	–
A_221	18/521	21/185	NV/525	NV/184	–	–
C_211A	–	–	–	–	–	–
C_211B	–	–	–	–	–	–
B_211	–	–	–	–	–	–
B_22	144/–	144/515	NV/–	NV/513	–	–
B_221	66/560	68/371	NV/749	NV/372	–	–
D_22	41/739	41/257	NV/740	NV/261	NV/768	NV/289

Table 23: RSA crypto key-length 1280 bit

Card	RSA (KeyLength 1536)					
	No Padding		PKCS#1 Pad		OAEP Pad	
	N*	C*	N*	C*	N	C
A_211	75/1297	74/410	NV/1297	NV/415	–	–
A_221	23/1226	24/291	NV/1228	NV/291	–	–
C_211A	–	–	–	–	–	–
C_211B	–	–	–	–	–	–
B_211	–	–	–	–	–	–
B_22	169/–	168/721	NV/–	NV/719	–	–
B_221	69/1226	69/544	NV/1214	NV/547	–	–
D_22	44/1238	44/409	NV/1233	NV/403	NV/1270	NV/440

Table 24: RSA crypto key-length 1536 bit

Card	RSA (KeyLength 1792)					
	No Padding		PKCS#1 Pad		OAEP Pad	
	N*	C*	N*	C*	N	C
A_211	128/6516	128/591	NV/6533	NV/591	–	–
A_221	25/1913	28/431	NV/1911	NV/431	–	–
C_211A	–	–	–	–	–	–
C_211B	–	–	–	–	–	–
B_211	–	–	–	–	–	–
B_22	192/–	190/980	NV/–	NV/981	–	–
B_221	68/1873	67/610	NV/1848	NV/609	–	–
D_22	55/1926	56/599	NV/1921	NV/599	NV/1967	NV/643

Table 25: RSA crypto key-length 1792 bit

Card	RSA (KeyLength 2048)					
	No Padding		PKCS#1 Pad		OAEP Pad	
	N*	C*	N*	C*	N	C
A_211	149/9656	151/833	NV/9646	NV/820	–	–
A_221	32/2820	32/614	NV/2822	NV/615	–	–
C_211A	–	–	–	–	–	–
C_211B	204/19087	209/509	NV/19008	NV/509	–	–
B_211	–	–	–	–	–	–
B_22	218/–	219/2825	NV/–	NV/2804	–	–
B_221	72/2605	CR	CR	CR	CR	CR
D_22	65/2827	65/857	NV/2826	NV/855	NV/2874	NV/901

Table 26: RSA crypto key-length 2048 bit

Card	RSA (KeyBuilder Class)							
	512*	768*	1024*	1280	1536	1792	2048	NSL
A_211	YES	YES	YES	YES	YES	YES	YES	YES^{+}
A_221	YES	YES	YES	YES	YES	YES	YES	YES
C_211A	YES	YES	YES	—	—	—	—	—
C_211B	—	—	YES	—	—	—	YES	—
B_211	YES	YES	YES	—	—	—	—	YES^{+}
B_22	YES	YES	YES	—	—	—	—	YES^{++}
B_221	YES	YES	YES	YES	YES	YES	YES	YES
D_22	YES	YES	YES	YES	YES	YES	YES	YES^{+}

Table 27: RSA key building

There are two categories of key types. The standard length keys and the non standard length keys. Visa Global Platform requires:

> `buildKey()` method shall support any other `keyLength` parameter (other than 512, 768 and 1024) for a `keyType`; `TYPE_RSA_CRT_PRIVATE`, `TYPE_RSA_PRIVATE` or `TYPE_RSA_PUBLIC`, as long as the length is between 512 and the maximum length supported by the card is 1024 bits and a multiple of 32 bits. Some cryptoprocessors support a maximum of 2048 bits but this is out of the scope of the Visa implementation.

The specification is unclear on key lengths between 1024 to 2048 bits. Nevertheless, all the key lengths have been tested.

The tables 27 and 28 give the results for the `KeyBuilder` class. Categories marked with * are required to be implemented for Visa compliance. $^{+}$ denotes that support for the non-standard key lengths was partial. $^{++}$ denotes that any non-standard key length less than or equal to 1024 was supported, else not supported. All key lengths are in bits, NSL means Non Standard Length, YES means Supported, — means Not Supported.

The DSA keys are supported only by the `B_22` card. It supports all the three key sizes (512 bits, 768 bits and 1024 bits).

Card	RSA CRT (KeyBuilder Class)							
	512*	768*	1024*	1280	1536	1792	2048	NSL
A_211	YES	YES	YES	YES	YES	YES	YES	YES$^+$
A_221	YES	YES	YES	YES	YES	YES	YES	NO
C_211A	YES	YES	YES	—	—	—	—	—
C_211B	—	—	YES	—	—	—	YES	—
B_211	YES	YES	YES	—	—	—	—	YES$^+$
B_22	YES	YES	YES	—	—	—	—	YES
B_221	YES	YES	YES	YES	YES	YES	YES	YES
D_22	YES	YES	YES	YES	YES	YES	YES	YES$^+$

Table 28: RSA CRT key building

Some of the cards support all the standard key lengths and the key lengths multiple of 32 bits. In the support of the non-standard lengths there is discrepancy. Some of the cards support up to 2048 bits, some support up to 1024 bits and some have random support. These have been indicated in the corresponding tables for both RSA and RSA CRT mode.

It was also observed that if the key size was set to be 0 bits, some of the cards like A_221 reset the key size to a default value and then continued with the rest of the testing. But any other key length not mentioned in the APIs or visa specifications returned an error.

The `buildKey` method allows applet to create a new key with the required specifications. Keys can then be set with methods like `setExponent` and `setModulus` and initialized. The methods differ from key to key. For e.g. for `RSAPublic` key we use `setModulus` and `setPublicExponent` and for `RSAPrivateKey` we use `setModulus` and `setPrivateExponent`.

A_211 This card supports all the standard key lengths in RSA as well as RSA CRT mode. It also supports non-standard key lengths, but not all of them.

A_221 This card supports all the standard key lengths in both RSA and RSA CRT mode. For non standard key lengths, this is more specific than the A_211 card. In the case of RSA, it supports all non-standard key lengths and for the case of RSA CRT mode, it does not support any non-standard key length.

C_211A None of the Manufacturer C cards support non standard key lengths. C_211A supports all the standard key lengths up to and including 1024 bits.

C_211B As mentioned, this card does not support non-standard key lengths. For the standard key lengths, it supports only 1024 and 2048 bits key lengths.

B_211 This card behaves similar to Manufacturer C cards except when dealing with non-standard key lengths. In standard key lengths, it supports everything up to and including 1024 bits. For non-standard key lengths it has partial support.

B_22 In RSA mode this card supports standard and non-standard key lengths up to and including 1024 bits. In RSA CRT mode, all key lengths, both standard and non-

Card	RSA (KeyGen)			
	512*	768*	1024*	>1024
A_211	—	—	—	—
A_221	—	—	—	—
C_211A	490	2555	4695	—
C_211B	—	—	—	—
B_211	—	—	—	—
B_22	2395	3555	6119	—
B_221	—	—	—	—
D_22	—	—	—	—

Table 29: Key-generation for RSA

Card	RSA CRT (KeyGen)						
	512*	768*	1024*	1280	1536	1792	2048
A_211	—	—	—	—	—	—	—
A_221	399	929	1425	5132	17228	25156	13559
C_211A	1044	1517	4696	—	—	—	—
C_211B	—	—	6688	—	—	—	20483
B_211	—	—	—	—	—	—	—
B_22	2531	3225	5679	8111	11757	—	—
B_221	—	—	—	—	—	—	—
D_22	1315	—	—	—	—	—	—

Table 30: Key-generation for RSA CRT

standard are supported. This is the only card that supports DSA key building. It supports all the three DSA key lengths (512, 768 and 1024 bits).

B_221 This card also has the maximum support. It supports key building of all key lengths for both RSA modes (RSA and RSA CRT). However, the card is unstable and crashed during encryption/decryption with 2048 bits key length.

D_22 This supports all standard key lengths in both RSA and RSA CRT mode. For non-standard key lengths, the support is partial.

3.6.4 On Card Key Generation

In this subsection, we discuss the ability of the card to generate keys. We test the on card key generation for different key lengths and draw a comparison between the various cards. The tables 29 and 30 give the complete results with the time taken to generate the keys wherever available.

In the previous subsection, we build empty keys and use the key information from the host to initialize the keys. Here we actually generate the keys on the card. Many card support building of the empty keys for use for a particular length of the keys, but do not support the generation of keys of that length. This explains why these two functionalities have been tested separately.

It was observed that for DSA, on card key generation was supported only by `B_22`. Only 512 bits key length is supported.

RSA key generation is more widely supported. Many key lengths are supported. RSA CRT is the most popular among all the supported algorithms. RSA in non-CRT mode is only supported by `C_211A` and `B_22` and only up to and including 1024 bits.

`A_211` This is one of the many cards that does not support any on-card key generation. Keys have to be loaded from outside.

`A_221` This is the only card to support key generation of all standard lengths. But this is only for RSA CRT mode. For RSA mode, it does not support key generation.

`C_211A` This card supports key generation in both RSA and RSA CRT mode but only up to and including key length 1024. This is the fastest card in both RSA and RSA CRT mode.

`C_211B` This card supports key generation in the RSA CRT mode. It is the slowest cards in terms of key generation times in the RSA CRT mode.

`B_211` Does not support any key generation.

`B_22` Supports key generation for smaller key lengths. In CRT mode it also supports a few larger ones. As already mentioned, this is the only card which supports DSA key generation, viz. for 512 bits.

`B_221` Does not support any key generation.

`D_22` Supports just one key generation in the CRT mode for the 512 bits key length.

3.6.5 Asymmetric Signature/Verification

Here the results from the testing of on card signature and verification have been tabulated and presented.

Signature The application initializes from the host by (a) generating, loading, and intializing the required keys on the applet, and (b) sending a text to be signed to the applet. The applet then generates a signature with the private key and then passes on the signatureto the host for verification. The host then generates his signature and compares it to the one received from the applet. If they match then the implementation is seen to be correct else it is seen to be incorrect. If the specified algorithm is not implemented on the card, a function not supported error is returned.

Verification The scenario follows the one above, but it's the applet that verifies (with the `verify()` method) the signature provided by the host and sends a yes/no answer to the host.

Card	RSA Sign/Verify							
	No Additional Padding						ISO9796	
	SHA1*		MD5		RMD160		SHA1*	
	N*	C*	N	C	N	C	N*	C*
A_211	477/112	205/VF	465/115	202/VF	SF/–	SF/–	577/41	315/42
A_221	297/17	125/VF	287/12	118/VF	SF/–	SF/–	327/44	156/44
C_211A	–	–	–	–	–	–	–	–
C_211B	–	–	–	–	–	–	–	–
B_211	473/101	325/VF	459/91	311/VF	SF/150	SF/VF	–	–
B_22	638/176	462/VF	600/154	418/VF	SF/187	SF/VF	–	–
B_221	625/162	305/VF	606/145	290/VF	SF/190	SF/VF	–	–
D_22	419/28	175/VF	411/24	171/VF	SF/–	SF/–	–	–

Table 31: RSA signing/verification, no padding and ISO9796

Card	RSA Sign/Verify									
	PKCS1						RFC2409			
	MD5		RMD160		SHA1		MD5		SHA1	
	N	C	N	C	N	C	N	C	N	C
A_211	697	431	–	–	–	–	–	–	–	–
A_221	430	258	–	–	–	–	–	–	–	–
C_211A	–	–	–	–	–	–	–	–	–	–
C_211B	–	–	–	–	–	–	–	–	–	–
B_211	795	644	691	543	793	645	–	–	677	523
B_22	979	794	780	600	923	735	–	–	–	–
B_221	–	–	823	509	–	–	–	–	856	542
D_22	503	265	–	–	–	–	555	315	–	–

Table 32: RSA signing/verification, PKCS1 and RFC2409

On the host side the algorithm implementations can be found in SunJCE in JDK 5.0 [14]. Some of the missing algorithms were found in Bouncy Castle Provider [3]. Few other algorithms were self implemented like signature algorithms with ISO9796 padding. None of the cards support DSA signatures. Also none of the cards supported RSA/RIPEMD160/ISO9796 or RSA CRT/RIPEMD160/ISO9796 combinations, so these results were not included in the tables.

The results are presented in Tables 31 and 32. From the results it is also apparent that both Manufacturer C cards do not support any of the signature algorithms for either RSA, RSA CRT mode or DSA. The time is indicated as Signature/Verification in milliseconds. Categories marked with * are required to be implemented for Visa compliance. N means RSA, C means RSA CRT, VF means Verification Failed, SF means Signature Failed,[10] — means Not Supported.

[10]Again, we find it really odd that, for example, all the cards failed in the same way for one algorithm (RMD160 with no padding). It may indicate that (a) the card indeed has a faulty implementation, or, more likely, (b) the signature algorithm combination we used on the host side is not the right one, i.e., not the one used on the card, despite all our best efforts to find the right one.

Another interesting observation during the testing was, the signature was correct in both RSA and RSA CRT mode for all cards, but verification failed in the RSA CRT mode, except for RSA/SHA1/ISO9796.

Table 32 shows the results for the algorithms which could not be implemented on the host side for verification. Their implementation could not be verified, but the card was checked to see if they are supported. If the applet successfully signs the data and verifies the signed data then the algorithm has been implemented for both generating a signature and verifying a signature, else the function is not supported and error code is returned. The times indicated in the table include both signing the data and then verifying the signed data.

`A_211` On all cards Signing and Verification are quicker in RSA CRT mode than in RSA mode. Though many of the algorithms are implemented on the card from Table 31, only one is implemented from Table 32. Speed of the card is average, neither quick nor slow.

`A_221` This is the fastest card for signing and verification. It is almost twice as quick. In terms of number of algorithms implemented both Manufacturer A cards are equal. Both Manufacturer A cards conform to the Visa Global Platform specs by implementing the ISO9796 padding with SHA1. They are the only cards to support this algorithm.

`C_211A`, `C_211B` None of the Manufacturer C cards support signature algorithms.

`B_211` This card has the most implementations. In Table 32 it supports all algorithms except RFC2409/MD5. Speed is comparable to the `A_211` card.

`B_22` This is the slowest card in terms of speed of operation. But it has fair number of implementations. Like all other cards, signing takes distinguishably longer than verification.

`B_221` Along with `B_22` card this is one of the slowest.

`D_22` Though the actual times are not very big, the ratio of signing time to verification time is the biggest for this card.

3.7 The RMI Applet

The RMI applet that we wrote has the same functionality as the `BasicInfo` applet, except that it uses the RMI interface. A suitable RMI host application has been written for the applet based on the IBM RMI off-card API. We successfully loaded and run the applet on all RMI supported cards: `A_221`, `B_22`, `B_221`, and `D_22`. One technical thing that was noticed on the way is that all cards except the `A_221` card require the CLA byte of RMI calls to be `80` hex, while the `A_221` card accepts both `00` and `80`. What we found nice (and surprising too to a certain extent) is that the IBM RMI off-card library can be used to talk to RMI applets on all cards. Having all the experiences with small differences in implementations on different cards and 'twists' in standard compliant solutions we found it really comforting.

3.8 Global and Open Platform API

The purpose of this test is to assess the card Global Platform API functionality and compliance to Visa GP specification. An applet[11] that utilizes the API has been written and a host application that communicates with the applet to extract the necessary information. The following things have been evaluated:

- Whether the card implements the API subset required by the VGP specification [17].
- Whether the Secure Channel and crypto related parts of the API give expected results. Namely, a secure channel is opened for the applet through the GP API (methods `processSecurity`, `openSecureChannel`, `verifyExternalAuthenticate`), and decryption routines are checked for accuracy according to the GP specification (not only whether they are supported, i.e., work giving *some* results) – methods `unwrap`, `decryptData`, and `decryptVerifyKey`.
- Additionally, the support for encryption (not required by VGP) have been tested – methods `wrap` and `encryptData`.
- Other parts of the API have been tested in a similar way by performing simple functionality tests. The exceptions are the `terminateCard` and `lockCard` methods. They were included in the source code of the applet in an unreachable block – we wanted to check whether the applet would link and load to the card. We did not want to actually invoke the methods and make cards unusable.
- The test applet is not loadable to the `C_211A` card. This is a known bug, which can be summarized as follows – due to a buggy (or 'too picky') on-card bytecode verifier, no applet that uses the OP API can be loaded onto the card. Certain versions of this card are not equipped with a bytecode verifier and thus are free from this problem. Unfortunately, all of our `C_211A` cards have the verifier enabled.

[11] Actually, because of major differences between the OP API 2.0.1 and GP API 2.1.1 two separate applets had to be written.

What we find *extremely disturbing* is that this bug is known since year 2003 and Manufacturer C still sells those cards to customers.

The results of this test can be summarized shortly:

- All the cards support the required API subset.
- For the GP 2.1.1 cards (`A_221` and `B_221`), all tests are passed successfully. However, none of the cards supports the encryption routines (`wrap` and `encryptData`). This seems natural as none of the cards supports R-MAC either.
- For the OP 2.0.1 cards (all the rest) almost all tests are passed successfully. The one thing we had problems with is setting and using the global PIN, for all but one card. Although it is claimed that all the cards support the global PIN, we were not able to set it through the API on most of the cards. The only OP 2.0.1 card that worked was the `D_22` card. We tried all different kinds of PIN formats and we always get the same problem – two cards return `false` on `setPin`, meaning that the PIN could not be set, two cards simply throw an exception with the error code 'wrong data'.

  We should also note that the global PIN API functionality for OP 2.0.1 API is very poorly treated in the publicly available documentation. The situation is slightly better in proprietary specifications, but that does not help much – despite our best effort (including setting the right privileges for the applet in the OP registry), we were not able to access the global PIN functionality on the OP 2.0.1 cards. Finally, we should note that global PIN worked fine on the GP 2.1.1 cards.
- Finally, this test gives us a final answer as to which OP/GP version the `D_22` card supports. The doubt came from the fact the information about this card found on the Internet suggest GP 2.1.1. However, both of our tests (the secure channel test and this one) indicate that the supported version is in fact OP 2.0.1.

3.9 Other APIs

The only other API that was open to testing on the card is the Java Card Forum Biometric API. Through the JCOP tool-set we have access to the Bio API 1.0 library files, so we made an attempt to put an applet that utilizes this API onto our cards. The test applet simply tries to instantiate all possible biometric templates defined by the API. The results of this test is the following: the test applet could only be loaded to two cards (`A_221` and `B_221`). For all the other cards the load failed, which means that (at least this particular version of) the Biometric API is not supported by the card. For the two cards that did accept the applet we got the following results: on the `B_221` card we could only instantiate a template associated with finger prints (`BioBuilder.FINGERPRINT`), on the `A_221` card we could not instantiate *any* of the templates. We find the latter result extremely strange, because it makes the whole API practically useless.[12] Thus, we are not sure if we can state

[12]Unless of course we missed something, but the fact that the same test worked for the `B_221` card suggests that something is not right here.

that the card supports the Biometric API, even though the API can be referenced on the card.

3.10 Preloaded Applets

We listed the load files, executable modules, applet AIDs, and security domains preexisting on each of the cards. The complete list is included in Appendix C in exactly the same form as our PATT tool provides. The listings include all extractable information from the GP registry, meaning, e.g., that if executable modules are not sub-listed for the load file that indicates that the card does not supply this information (in fact, only two cards do – the `A_221` and `D_22`). The `D_22` card caused a little trouble when retrieving the load file listing: either the only `D_22` card we have left has a corrupted registry because of one of our other tests, or the card simply reports the load file information in a wrong way. In either case the card reports executable modules for one load file correctly, and for the other load file the executable modules data is simply malformed (we checked "manually" – one byte required by the GP specification is missing in card's response).

We leave it to reader's discretion to relate the listed AIDs to any of the smart card software or libraries known in the industry.

3.11 Garbage Collection/Memory Management

The garbage collection and memory management test was performed in two steps. In the first step the Java/applet memory was tested for garbage collection.[13] To this end an applet has been written that performs the following tests:

- Subsequent allocation of memory blocks storing the references to the newly allocated memory. In this case the applet should always (eventually) run out of memory (the amount of currently allocated memory is noted down when this happens – the results given below are approximate amounts, because the memory is allocated in blocks, thus, when it is not possible to allocate a new whole block it does not necessarily mean there the amount of memory left is 0). After the memory is filled, all the references are 'released' and the test is rerun. At this point the applet is expected to be able to allocate memory again, if the garbage collection is implemented.

- Subsequent allocation and immediate release of memory blocks. If the garbage collection is fully implemented it should be possible to do it indefinitely (a certain limit is put into the test to stop after we are sure that the test is passed). There are two flavours of this test – one that stores the reference to the newly allocated block in an applet field and one that stores the same reference in a local variable. This is done

[13] As strongly noted in the Java Card documentation, in Java smart cards garbage collection mechanism does not exist as such, i.e., the one known from the desktop Java. On Java Cards the mechanism is called *object deletion* and on new Java Cards has to be invoked explicitly through an API call. By default, older cards do not support garbage collection and/or object deletion.

in the hope that some JCVMs may treat references differently depending on where they are stored (persistent or transient memory).

For all of the above tests both kinds of memory were tested – RAM and EEPROM, i.e., the memory blocks were allocated in either. Additionally, for the cards that support Java Card API 2.2.*, we checked if the method `requestObjectDeletion` has to be invoked for the garbage collection to work (turns out all the JC 2.2.* cards require this call to collect garbage). Moreover, since the JC API 2.2 can report on free memory on the card, we compared the results reported by the API with what the expectations were according to our test data. For the transient (RAM) memory interesting results were obtained.

The following is the description of each card behaviour:

`A_221` The garbage collection seems to be fully implemented on this card. The total amount of persistent memory we managed to allocate was in the vicinity of 68KB.[14] An interesting thing we discovered about RAM allocation is the following: When allocating memory in the 'clear on deselect' mode it seems that the whole available RAM can be allocated (circa 1900 bytes), but the API does not report *any* change in transient memory size after allocation. Conversely, when allocating in the 'clear on reset' mode, the API correctly reports the new memory size, but the whole available memory cannot be allocated. That would suggest that the JCVM keeps a reserve block (around 200 bytes) of RAM for whatever purposes.

`B_221` This card also fully implements garbage collection. The amount of persistent memory that we managed to allocate was in the vicinity of 47KB. When it comes to RAM allocation and API reports of the available memory size, the card is a kind of mystery. In short: the memory is allocated correctly and the maximum allocatable sizes are what is expected: around 900 bytes can be allocated in the 'clear on deselect' mode, slightly less (600 bytes) in the 'clear on reset' mode. It seems that similar limitations on the 'clear on reset' mode apply as for the `A_221` card. However, the reports that the API gives do follow a pattern, but one very difficult to explain. We noted earlier that this card reports two different memory sizes for the two allocation modes ('reset' and 'deselect'). When we allocate the 'deselect' memory, the card report less 'reset' memory available, but not accordingly to what was allocated. When we allocate the 'reset' memory, the two types of memory are reported to be less according to the amount we allocated. Apparently the book keeping is not very accurate, similar to the `A_221` card.

`B_22` In principle the card exhibits the same behaviour as the `A_221` card – garbage collection is fully implemented, the amount of persistent memory we managed to allocate was circa 56KB. Reporting of free transient memory works almost the same way as on `A_221` card – the card does not properly keep track of 'clear on deselect' allocations. The difference is that on this card the whole transient memory (circa 1900 bytes) can be allocated in both modes – the card does not keep any reserves.

[14] Note again, that none of the cards during this test were 'empty', thus the reported memory sizes should be taken with certain reserve.

D_22 Again, the garbage collection fully works on this card. The amount of persistent memory that we managed to allocate was 55KB (there was one big applet that could not be removed installed on the card during the test). The amount of allocatable transient memory was in the range of 900 bytes in both allocation modes. Similar to other cards, this card has some problems reporting free transient memory – after allocating the memory in the 'reset' mode the reports are as expected, after allocation in the 'deselect' mode only the 'reset' transient memory is reported decreased, the 'deselect' memory is reported as it was before the allocation.

The rest of the cards do not support Java Card API 2.2 or higher and thus are not capable of reporting the available memory through API calls. Also object deletion is not supported through the API on these cards, nevertheless we tested them all to see if the cards *possibly* have any implicit garbage collection mechanism.

A_211 There is no application level garbage collection on this card. We managed to allocate circa 28KB of the persistent memory on this card, circa 500 bytes of RAM in the 'reset' mode and circa 600 bytes in the 'deselect' mode.

B_211 Same as A_211 card. We managed to allocate circa 14KB of the persistent memory on this card, circa 2100 bytes of RAM in the "reset" mode and circa 2300 bytes in the "deselect" mode.

C_211A, C_211B Also no application level garbage collection on these cards. We managed to allocate circa 28KB of persistent memory, and circa 700 bytes of transient memory (in both allocation modes) on C_211A, and 1000 bytes (in both allocation modes) on C_211B. A note w.r.t. allocatable persistent memory is due here: Both Manufacturer C cards require the user to specify how much persistent memory the applet is allowed to use. On both cards we specified this to be around 29KB (70FF hex), giving the maximum specifiable value of 32KB (7FFF hex) is not possible – the applet is refused installation. Thus, it is not surprising that neither of the cards (even though the newer one is supposedly 64K) allowed us to allocate more than 29KB.

In parallel to the applet level garbage collection test we performed the card registry memory management test, i.e., we tested whether applet deletion takes proper care of any garbage that the applet may be leaving behind. Whenever we filled the memory with garbage by running the test applet described above, we deleted the applet, reinstalled it, and checked whether we can allocate the same amount of memory as before. This cycle was repeated a sufficiently large number of times to convince us that the garbage is indeed cleaned up upon applet deletion. All the cards successfully passed this test. At this point we also have to revise some statements we made earlier about the C_211A card. One instance of this card did fail on us w.r.t. memory management (see Section 3.2) and during key management test (see Section 3.3.1). It seems though that none of the failures were caused by bugs in the card or lack of garbage collection or other features. To repeat what we have said earlier, probably what happened is that we introduced some faults into the cards in the earlier stages of testing (the transaction test could be responsible, as it,

e.g., totally destroyed the D_22 card). Unfortunately, we are not able to back trace these faults.

The overall result of the garbage collection test can be summarized as follows. None of the older generation cards (JC API 2.1.1) support garbage collection on the applet level. All of the new generation cards (JC API 2.2 and on) fully support garbage collection, or more precisely, object deletion. An adequate API call is required on those cards to trigger the garbage collection mechanism. This seems reasonable, as garbage collecting takes visibly long time and is very expensive on EEPROM memory writes.

3.12 Software and Test Results Availability

All software and applets written for the tests as well as log files (where applicable) from the tests are available on request. Some of the software (not always up to date) is available on the our web page: `http://www.cs.ru.nl/~woj/software/software.html`.

4 Conclusions

The outcome of the study we presented in this report has multiple aspects:

- The test results themselves – only after the first stage of testing it was already clear that the cards differ substantially from each other in small details. Towards the end of the testing a number of card differentiators have been discovered.

- Broken cards – it is surprising to see how easy it is to break some of the cards. One of the D_22 cards did not even last 30 minutes of testing, and we did loose one of our C_211A cards on the way (in a sense it still functions, but is practically useless as a test card). We think that when it comes to choosing cards for long term use, the resilience aspect is probably one of the most important ones. Also, we cannot imagine, for example, how contactless cards with a faulty transaction mechanism can survive the 'large market' reality.

- Loopholes in specifications – testing of cards revealed that there are some unexplained issues in different kinds of specifications. One example is all the ambiguities in the transaction mechanism specification that are being progressively corrected, the other one is the behaviour of Cipher for inputs of length zero, which seems to be underspecified.

- The (steep) learning curve – most of the time spent in the project was devoted to finding suitable information, studying all different standards involved in the Java Card technology, and getting acquainted with the different API libraries. Although lots of time has been consumed, we think it paid off, both in terms of the test results and knowledge gained.

- Software – finally, lots of host-side applications and card applets have been developed in the process. We are planning on developing them further and making them publicly available as much as possible.

References

[1] Axalto Smart Card Security. *Cyberflex access 64k FIPS140-2 Level 3 Cryptographic Module Security Policy.* `http://csrc.nist.gov/cryptval/140-1/140sp/140sp572.pdf`.

[2] Axalto Smart Card Security. *Cyberflex e-gate 32k Techinical Specifications.* `http://www.cyberflex.com/Products/cards_egate.html`.

[3] *Bouncy Castle Crypto APIs*, October 2006. `http://www.bouncycastle.org/docs/docs1.5/index.html`.

[4] J. Castella, J. Domingo-Ferrer, J. Herrera, and J. Planes. A performance comparison of Java Cards for micropayment implementation, 2000. Compares GemXpresso JC2.0(Gemplus), Odyssey JC2.1(Bull), SmardCafe JC2.1(Giesecke & Devrient),Cyberflex 2.a(Schlumberger).

[5] EMV. *EMV Card Personalization Specification*, September 2006. `http://www.emvco.com/specifications.asp`.

[6] Giesecke & Devrient GmbH. *Sm@rtCafé Expert 64 The Java based smart card.* `http://www.gi-de.com/pls/portal/maia.display_custom_items.DOWNLOAD_SEEALSO_FILE?p_ID=5511&p_page_id=86690&p_pg_id=42`.

[7] Global Platform Organization. *Card Specification, Version 2.1.1*, March 2003. `http://www.globalplaform.org`.

[8] E. Hubbers, W. Mostowski, and E. Poll. Tearing Java Cards. In *Proceedings, e-Smart 2006, Sophia-Antipolis, France, September 20–22*, 2006. This paper was one of three papers preselected for the Java Card Forum Contest held at e-Smart 2006.

[9] E. Hubbers and E. Poll. Transactions and non-atomic API calls in Java Card: Specification ambiguity and strange implementation behaviours. Department of Computer Science NIII-R0438, Radboud University Nijmegen, 2004.

[10] International Standard Organization. *ISO/IEC 9797-1:1999(E) - Infomation technology - Secruity techniques - Message Authentication Codes (MACs)*, December 1999.

[11] *Java Card 2.1.1 Application Programming Interface*, May 2000. `http://java.sun.com/products/javacard/specs.html`.

[12] *Java Card 2.2 Application Programming Interface*, September 2002. http://java.sun.com/products/javacard/specs.html.

[13] *Application Programming Interface, Java Card Platform, Version 2.2.1*, October 2003. http://java.sun.com/products/javacard/specs.html.

[14] *Java Cryptography Extention (JCE) Reference Guide for the Java 2 Platform Standard Edition Development Kit (JDK) 5.0*, 2005. http://java.sun.com/j2se/1.5.0/docs/guide/security/jce/JCERefGuide.html.

[15] Philips Semiconductors. *Philips P8WE5033 Secure 8-bit Smart Card Controller Short Form Specification*, June 2001. http://www.semiconductors.philips.com/acrobat_download/other/identification/sfs051411.pdf.

[16] Philips Semiconductors. *Philips SmartMX P5CT072 Secure Dual Interface PKI Smart Card Controller Short Form Specification*, October 2004. http://www.semiconductors.philips.com/acrobat/other/identification/sfs085513.pdf.

[17] Visa. *Visa Global Platform 2.1.1 Card Implementation Requirements*, May 2003. http://partnernetwork.visa.com/cd/globalplat/main.jsp.

A BasicInfo Test Results

- B_211:

```
Card reports supported baud rate of 53763
API version:      2.1
APDU protocol:    Type T1, media DEFAULT
Max Commit:       505
APDU buffersize:  261
```

- A_211 (2.1.1):

```
Card reports supported baud rate of 9622 (default)
API version:      2.1
APDU protocol:    Type T0, media DEFAULT
Max Commit:       500
APDU buffersize:  261
```

- C_211A:

```
Card reports supported baud rate of 78125
API version:      2.1
APDU protocol:    Type T0, media DEFAULT
Max Commit:       510
APDU buffersize:  261
```

- C_211B:

```
Card reports supported baud rate of 156250
API version:      2.1
APDU protocol:    Type T0, media DEFAULT
Max Commit:       512
APDU buffersize:  272
```

- A_221:

```
Card reports supported baud rate of 161290
  (Regular A_221 in contact mode)
Card reports supported baud rate of 9622 (default)
  (Regular A_221 in contactless mode,
  the other flavour of A_221 in both modes)
API version:               2.2
APDU protocol:             Type T1, media DEFAULT  (contact)
APDU protocol:             Type TCL, media DEFAULT  (contactless)
Max Commit:                512
Free Commit:               512
Free Memory Persistent:    32767
Free Memory TransReset:    1983
Free Memory TransDeselect: 1983
Object Deletion supported: Yes
APDU buffersize:           261
Default RMI INS:           0x38
```

- B_22:

```
Card reports supported baud rate of 161290
API version:               2.2
APDU protocol:             Type T1, media DEFAULT
Max Commit:                896
Free Commit:               896
Free Memory Persistent:    32767
Free Memory TransReset:    1936
Free Memory TransDeselect: 1936
Object Deletion supported: Yes
APDU buffersize:           261
Default RMI INS:           0x38
```

- B_221:

```
Card reports supported baud rate of 161290         (contact)
Card reports supported baud rate of 9622 (default) (contactless)
API version:               2.2
APDU protocol:             Type T1, media DEFAULT (contact)
APDU protocol:             Type TCL, media DEFAULT (contactless)
```

```
Max Commit:                504
Free Commit:               504
Free Memory Persistent:    32767
Free Memory TransReset:    614
Free Memory TransDeselect: 906
Object Deletion supported: Yes
APDU buffersize:           261
Default RMI INS:           0x38
```

- D_22:

```
Card reports supported baud rate of 161290         (contact)
Card reports supported baud rate of 9622 (default) (contactless)
API version:               2.2
APDU protocol:             Type T0, media DEFAULT (contact)
APDU protocol:             Type TCL, media DEFAULT (contactless)
Max Commit:                511
Free Commit:               511
Free Memory Persistent:    32767
Free Memory TransReset:    946
Free Memory TransDeselect: 946
Object Deletion supported: Yes
APDU buffersize:           274
Default RMI INS:           0x38
```

B Global Platform Test Results

- B_211:

```
The main Secure Channel Protocol is: 1
OP/GP version most likely is: 2.0.1
Supported Secure Channel Versions are:
  [SCP_01_05 APDU_CLR APDU_MAC APDU_ENC]
Key deletion supported: Yes
VISA GP predictable challenge: No
Number of supplementary logical channels supported: 0
Additional Security Domain instantiated successfully
```

- A_211 (2.1.1):

```
The main Secure Channel Protocol is: 1
OP/GP version most likely is: 2.0.1
Supported Secure Channel Versions are:
  [SCP_01_05 APDU_CLR APDU_MAC APDU_ENC]
Key deletion supported: No
VISA GP predictable challenge: No
Number of supplementary logical channels supported: 0
No default load file found to instantiate additional Security Domain
```

- C_211A:

```
The main Secure Channel Protocol is: 1
OP/GP version most likely is: 2.0.1
Supported Secure Channel Versions are:
  [SCP_01_05 APDU_CLR APDU_MAC APDU_ENC]
Key deletion not tested, card does very strange things
  with key registry!
Key deletion supported: No
VISA GP predictable challenge: No
Number of supplementary logical channels supported: 0
No default load file found to instantiate additional Security Domain
```

- C_211B:

```
The main Secure Channel Protocol is: 1
OP/GP version most likely is: 2.0.1
Supported Secure Channel Versions are:
  [SCP_01_05 APDU_CLR APDU_MAC APDU_ENC]
Key deletion supported: No
VISA GP predictable challenge: No
Number of supplementary logical channels supported: 0
Could not instantiate additional Security Domain
```

- A_221:

```
The main Secure Channel Protocol is: 2
OP/GP version most likely is: 2.1.1
Supported Secure Channel Versions are:
  [SCP_02_15 APDU_CLR APDU_MAC APDU_ENC]
Key deletion supported: No
VISA GP predictable challenge: No
Number of supplementary logical channels supported: 3
Logical Channels: Multi-selection of Security Domain not possible.
Additional Security Domain instantiated successfully
```

- B_22:

```
The main Secure Channel Protocol is: 1
OP/GP version most likely is: 2.0.1
Supported Secure Channel Versions are:
  [SCP_01_05 APDU_CLR APDU_MAC APDU_ENC]
Key deletion supported: Yes
VISA GP predictable challenge: No
Number of supplementary logical channels supported: 3
Logical Channels: Multi-selection of Security Domain possible.
  Opening of a secure channel on sup. logical channel possible.
  Communication over multiple secure/logical channels failed,
  only one secure/logical channel active.
Additional Security Domain instantiated successfully
```

- B_221:

```
The main Secure Channel Protocol is: 2
OP/GP version most likely is: 2.1.1
Supported Secure Channel Versions are:
  [SCP_02_15 APDU_CLR APDU_MAC APDU_ENC]
Key deletion supported: No
VISA GP predictable challenge: No
Number of supplementary logical channels supported: 3
Logical Channels: Multi-selection of Security Domain possible.
  Opening of a secure channel on sup. logical channel possible.
  Communication over multiple secure/logical channels failed,
  only one secure/logical channel active.
Additional Security Domain instantiated successfully
```

- D_22:

```
The main Secure Channel Protocol is: 1
OP/GP version most likely is: 2.0.1
Supported Secure Channel Versions are:
  [SCP_01_05 APDU_CLR APDU_MAC APDU_ENC]
Key deletion supported: Yes
VISA GP predictable challenge: No
Number of supplementary logical channels supported: 3
Logical Channels: Multi-selection of Security Domain not possible.
Could not instantiate additional Security Domain
```

C Preinstalled Applets

- B_211:

```
Card Manager (ISD) status:
AID: A0 00 00 00 03 00 00 00
State: READY

Security Domain status:
-------------------------------------------
None found.
-------------------------------------------

Applet status:
-------------------------------------------
None found.
-------------------------------------------

Load File status:
-------------------------------------------
```

```
AID: A0 00 00 00 03 53 50  [?????SP]
State: LOADED
Executable modules:
    Not available.
-----------------------------------------
AID: A0 00 00 00 03 53 44  [?????SD]
State: LOADED
Executable modules:
    Not available.
-----------------------------------------
AID: D2 76 00 00 05 AA 04 03 60 01 04  [?v??????‘??]
State: LOADED
Executable modules:
    Not available.
-----------------------------------------
AID: D2 76 00 00 05 AA FF CA FE 00 01  [?v??????????]
State: LOADED
Executable modules:
    Not available.
-----------------------------------------
AID: A0 00 00 00 03 00 00  [???????]
State: LOADED
Executable modules:
    Not available.
-----------------------------------------
AID: A0 00 00 00 62 02 01  [????b??]
State: LOADED
Executable modules:
    Not available.
-----------------------------------------
AID: A0 00 00 00 62 01 02  [????b??]
State: LOADED
Executable modules:
    Not available.
-----------------------------------------
AID: A0 00 00 00 62 01 01  [????b??]
State: LOADED
Executable modules:
    Not available.
-----------------------------------------
AID: A0 00 00 00 62 00 01  [????b??]
State: LOADED
Executable modules:
    Not available.
-----------------------------------------
```

- A_211 (2.1.1):

```
Card Manager (ISD) status:
```

```
AID: A0 00 00 00 03 00 00 00
State: INITIALIZED

Security Domain status:
-----------------------------------------
None found.
-----------------------------------------

Applet status:
-----------------------------------------
None found.
-----------------------------------------

Load File status:
-----------------------------------------
AID: A0 00 00 00 62 00 01  [????b??]
State: LOADED
Executable modules:
    Not available.
-----------------------------------------
AID: A0 00 00 00 62 01 01  [????b??]
State: LOADED
Executable modules:
    Not available.
-----------------------------------------
AID: A0 00 00 00 62 01 02  [????b??]
State: LOADED
Executable modules:
    Not available.
-----------------------------------------
AID: A0 00 00 00 62 02 01  [????b??]
State: LOADED
Executable modules:
    Not available.
-----------------------------------------
AID: A0 00 00 00 03 00 00  [???????]
State: LOADED
Executable modules:
    Not available.
-----------------------------------------
AID: 31 50 41 59 2E  [1PAY.]
State: LOADED
Executable modules:
    Not available.
-----------------------------------------
AID: A0 00 00 00 03 60 10  [?????‘?]
State: LOADED
Executable modules:
    Not available.
```

```
-------------------------------------------
```

- C_211A:

```
Card Manager (ISD) status:
AID: A0 00 00 00 03 00 00
State: INITIALIZED

Security Domain status:
-------------------------------------------
None found.
-------------------------------------------

Applet status:
-------------------------------------------
None found.
-------------------------------------------

Load File status:
-------------------------------------------
AID: A0 00 00 00 62 00 01  [????b??]
State: LOADED
Executable modules:
    Not available.
-------------------------------------------
AID: A0 00 00 00 62 01 01  [????b??]
State: LOADED
Executable modules:
    Not available.
-------------------------------------------
AID: A0 00 00 00 62 01 02  [????b??]
State: LOADED
Executable modules:
    Not available.
-------------------------------------------
AID: A0 00 00 00 62 02 01  [????b??]
State: LOADED
Executable modules:
    Not available.
-------------------------------------------
AID: A0 00 00 00 03 00 00  [???????]
State: LOADED
Executable modules:
    Not available.
-------------------------------------------
```

- C_211B:

```
Card Manager (ISD) status:
```

```
AID: A0 00 00 00 03 00 00 00
State: INITIALIZED

Security Domain status:
-------------------------------------------
None found.
-------------------------------------------

Applet status:
-------------------------------------------
None found.
-------------------------------------------

Load File status:
-------------------------------------------
AID: A0 00 00 00 62 00 01  [????b??]
State: LOADED
Executable modules:
    Not available.
-------------------------------------------
AID: A0 00 00 00 62 01 01  [????b??]
State: LOADED
Executable modules:
    Not available.
-------------------------------------------
AID: A0 00 00 00 62 01 02  [????b??]
State: LOADED
Executable modules:
    Not available.
-------------------------------------------
AID: A0 00 00 00 62 02 01  [????b??]
State: LOADED
Executable modules:
    Not available.
-------------------------------------------
AID: A0 00 00 00 03 00 00  [???????]
State: LOADED
Executable modules:
    Not available.
-------------------------------------------
AID: A0 00 00 00 30 00 00 70 00 68 00 10 20 30  [????0??p?h?? 0]
State: LOADED
Executable modules:
    Not available.
-------------------------------------------
AID: A0 00 00 01 32 00 01  [?????2??]
State: LOADED
Executable modules:
    Not available.
```

```
-------------------------------------------
AID: A0 00 00 00 30 29 05 70 00 AD 14 10 01 01  [????0)?p??????]
State: LOADED
Executable modules:
    Not available.
-------------------------------------------
AID: A0 00 00 00 03 53 44  [?????SD]
State: LOADED
Executable modules:
    Not available.
-------------------------------------------
AID: A0 00 00 00 03 53 50  [?????SP]
State: LOADED
Executable modules:
    Not available.
-------------------------------------------
```

- A_221:

```
Card Manager (ISD) status:
AID: A0 00 00 00 03 00 00 00
State: READY

Security Domain status:
-------------------------------------------
None found.
-------------------------------------------

Applet status:
-------------------------------------------
None found.
-------------------------------------------

Load File status:
-------------------------------------------
AID: A0 00 00 00 03 53 50  [?????SP]
State: LOADED
Executable modules:
    A0 00 00 00 03 53 50 41 [?????SPA]
-------------------------------------------
```

- B_22:

```
Card Manager (ISD) status:
AID: A0 00 00 00 03 00 00 00
State: INITIALIZED

Security Domain status:
```

```
-----------------------------------------
None found.
-----------------------------------------

Applet status:
-----------------------------------------
None found.
-----------------------------------------

Load File status:
-----------------------------------------
AID: D2 76 00 00 05 AA FF CA FE 00 01  [?v??????????]
State: LOADED
Executable modules:
    Not available.
-----------------------------------------
AID: A0 00 00 00 62 00 01  [????b??]
State: LOADED
Executable modules:
    Not available.
-----------------------------------------
AID: A0 00 00 00 62 00 02  [????b??]
State: LOADED
Executable modules:
    Not available.
-----------------------------------------
AID: A0 00 00 00 62 00 03  [????b??]
State: LOADED
Executable modules:
    Not available.
-----------------------------------------
AID: A0 00 00 00 62 01 01  [????b??]
State: LOADED
Executable modules:
    Not available.
-----------------------------------------
AID: A0 00 00 00 62 01 01 01  [????b???]
State: LOADED
Executable modules:
    Not available.
-----------------------------------------
AID: A0 00 00 00 62 01 02  [????b??]
State: LOADED
Executable modules:
    Not available.
-----------------------------------------
AID: A0 00 00 00 62 02 01  [????b??]
State: LOADED
Executable modules:
```

```
    Not available.
-------------------------------------------
AID: A0 00 00 01 51 00  [????Q?]
State: LOADED
Executable modules:
    Not available.
-------------------------------------------
AID: A0 00 00 00 03 00 00  [???????]
State: LOADED
Executable modules:
    Not available.
-------------------------------------------
AID: D2 76 00 00 05 AA 04 03 60 01 04  [?v??????‘??]
State: LOADED
Executable modules:
    Not available.
-------------------------------------------
AID: A0 00 00 00 03 53 44  [?????SD]
State: LOADED
Executable modules:
    Not available.
-------------------------------------------
AID: A0 00 00 00 03 53 50  [?????SP]
State: LOADED
Executable modules:
    Not available.
-------------------------------------------
```

- B_221:

```
Card Manager (ISD) status:
AID: A0 00 00 00 03 00 00 00
State: INITIALIZED

Security Domain status:
-------------------------------------------
None found.
-------------------------------------------

Applet status:
-------------------------------------------
AID: A0 00 00 00 63 50 4B 43 53 2D 31 35  [????cPKCS-15]
State: SELECTABLE
Privileges: []
-------------------------------------------

Load File status:
-------------------------------------------
AID: A0 00 00 00 03 53 44  [?????SD]
```

```
State: LOADED
Executable modules:
    Not available.
-----------------------------------------
AID: A0 00 00 00 03 53 50  [?????SP]
State: LOADED
Executable modules:
    Not available.
-----------------------------------------
AID: A0 00 00 02 27 01 10 00  [????'???]
State: LOADED
Executable modules:
    Not available.
-----------------------------------------
AID: 53 4B 54 45 58 54 4C 49 42  [SKTEXTLIB]
State: LOADED
Executable modules:
    Not available.
-----------------------------------------
AID: A0 00 00 00 63 02  [????c?]
State: LOADED
Executable modules:
    Not available.
-----------------------------------------
```

- D_22:

```
Card Manager (ISD) status:
AID: A0 00 00 00 03 00 00 00
State: READY

Security Domain status:
-----------------------------------------
None found.
-----------------------------------------

Applet status:
-----------------------------------------
None found.
-----------------------------------------

Load File status:
-----------------------------------------
AID: A0 00 00 00 03 53 50  [?????SP]
State: LOADED
Executable modules:
    A0 00 00 01 51 00 00 [????Q??]
    A0 00 00 00 03 53 50 41 [?????SPA]
    A0 00 00 00 03 00 00 [???????]
```

```
-------------------------------------------
AID: A0 00 00 00 77 01 00 03 00 10 00 00 00 00 00 03  [????w???????????]
State: LOADED
Executable modules:
    Not available.
-------------------------------------------
```