Evaluation of Java Card Performance

Samia Bouzefrane[1], Julien Cordry[1],
Hervé Meunier[2], and Pierre Paradinas[3]

[1] CNAM 292 rue Saint-Martin 75003 Paris France
firstname.lastname@cnam.fr
[2] INRIA POPS Parc Scientifique de la Haute Borne Bt. IRCICA 50,
avenue Halley - BP 70478 59658 Villeneuve d'Ascq, France
herve.meunier@lifl.fr
[3] INRIA Rocquencourt 78150 Le Chesnay France
Pierre.Paradinas@inria.fr

Abstract. With the growing acceptance of the Java Card standard, understanding the performance behaviour of these platforms is becoming crucial. To meet this need, we present in this paper, a benchmark framework that enables performance evaluation at the bytecode and API levels. We also show, how we assign, from the measurements, a global mark to characterise the efficiency of a given Java Card platform, and to determine its performance according to distinct smart card profiles.

Keywords: Java Card, Benchmark, Performance, Test.

1 Introduction

The advent of the Java Card standard has been a major turning point in smart card technology. It provides a secure, vendor-independent, ubiquitous Java platform for smart cards. It shortens the time-to-market and enables programmers to develop smart card applications for a wide variety of vendors' products.

In this context, understanding the performance behaviour of Java Card platforms is important to the Java Card community. Currently, there is no solution on the market which makes it possible to evaluate the performance of a smart card that implements Java Card technology. In fact, the programs which realize this type of evaluations are generally proprietary and not made available to the whole of the Java Card community. Hence, the only existing and published benchmarks are used within research laboratories (e.g., SCCB project from CEDRIC laboratory [3,6], or IBM Research [11]). However, benchmarks are important in the smart card area because they contribute in discriminating companies products, especially when the products are standardised.

Our purpose is to describe the different steps necessary to measure the performance of the Java Card platforms. In this paper, the emphasis is towards determining the optimal parameters to enable measurements that are as accurate and linear as possible. We also show, how we assign, from the measurements, a global mark to characterise the efficiency of a given Java Card platform, and to determine its performance according to distinct smart card profiles.

G. Grimaud and F.-X. Standaert (Eds.): CARDIS 2008, LNCS 5189, pp. 228–240, 2008.

The remainder of this paper is organised as follows. In Section 2, we describe the Java Card technology. Subsequently, we detail in Section 3 the different modules that compose the framework architecture. Section 4 presents a state of the art of the benchmarking attempts in smart card area before concluding the paper in Section 5.

2 Java Card and Benchmarking

2.1 Java Card Technology

Java Card technology provides means of programming smart cards [2,8] with a subset of the Java programming language. Today's smart cards are small computers, providing 8, 16 or 32 bits CPU with clock speeds ranging from 5 up to 40MHz, ROM memory between 32 and 128KB, EEPROM memory (writable, persistent) between 16 and 64KB and RAM memory (writable, non-persistent) between 3 and 5KB. Smart cards communicate with the rest of the world through application protocol data units (APDUs, ISO 7816-4 standard). The communication is done in master-slave mode. It is always the terminal application that initialises the communication by sending the command APDU to the card and then the card replies by sending a response APDU (possibly with empty contents). In the case of Java powered smart cards, the cards ROM contains, in addition to the operating system, a Java Card Virtual Machine (JCVM), which implements a subset of the Java programming language, hence allowing Java Card applets to run on the card.

A Java Card applet should implement the `install` method responsible for initializing the applet (usually calle by the applet constructor) and a `process` method for handling incoming command APDUs and sending the response APDUs back to the host. More than one applet can be installed on a single card, however only one can be active at a time (the active one is the most recently selected by the Java Card Runtime Environment – JCRE). A normal Java compiler is used to convert the source code into Java bytecodes. Then a converter must be used to convert the bytecode into a more condensed form (CAP format) that can be loaded onto a smart card. The converter also checks that no unsupported features (like floats, strings, etc.) are used in the bytecode. This is sometimes called off-card or off-line bytecode verification.

2.2 Addressed Issues

Our research work falls under the MESURE project [12], a project funded by the French administration (ANR[1]), which aims at developing a set of open source tools to measure the performance of Java Card platforms.

Only features related to the normal use phase of Java Card applications will be considered here. Are excluded features like installing, personalizing or deleting

[1] `http://www.agence-nationale-recherche.fr/`

an application since they are of lesser importance from user's point of view and performed once.

Hence, the benchmark framework enables performance evaluation at three levels:

- The VM level: to measure the execution time of the various instructions of the virtual machine (basic instructions), as well as subjacent mechanisms of the virtual machine (e.g., reading and writing the memory).
- The API level: to evaluate the functioning of the services proposed by the libraries available in the embedded system (various methods of the API, namely those of Java Card and GlobalPlatform).
- The JCRE level: to evaluate the non-functional services, such as the transaction management, the method invocation in the applets, etc.

The set of tests are supplied to benchmark Java Card platforms available for anybody and supported by any card reader. The various tests thus have to return accurate results, even if they are not executed on precision readers. We reach this goal by removing the potential card reader weakness (in terms of delay, variance and predictability) and by controlling the noise generated by measurement equipment (the card reader and the workstation). Removing the noise added to a specific measurement can be done with the computation of an average value extracted from multiple samples. As a consequence, it is important on the one hand to perform each test several times and to use basic statistical calculations to filter the trustworthy results. On the other hand, it is necessary to execute several times in each test the operation to be measured in order to fix a minimal duration for the tests (> 1 second) and to expect getting precise results.

We will not take care of features like the I/Os or the power consumption because their measurability raises some problems such as:

- For a given smart card, distinct card readers may provide different I/Os measurements.
- Each part of an APDU is managed differently on a smart card reader. The 5 bytes header is read first, and the following data can be transmitted in several way: 1 acknowledge for each byte or not, delay or not before noticing the status word.
- The smart card driver used by the workstation generally induces more delay on the measurement than the smart card reader itself.

3 General Benchmarking Framework

3.1 Introduction

We defined a set of modules as part of the benchmarking framework. The general framework is illustrated in the figure 1.

The benchmarks have been developed under the Eclipse environment based on JDK 1.6, with JSR268. The underlying ISO 7816 smart card architecture forces us

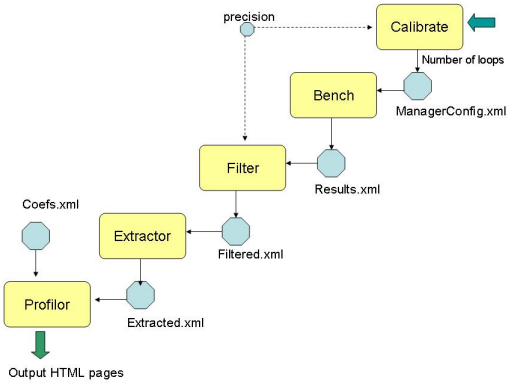

Fig. 1. Overall Architecture

to measure the time a Java Card platform takes to answer to a command APDU, and to use that measure to deduce the execution time of some operations.

The benchmarking development tool covers two parts: the script part and the applet part. The script part, entirely written in Java, defines an abstract class that is used as a template to derive test cases characterized by relevant measuring parameters such as, the operation type to measure, the number of loops, etc. A method `run()` is executed in each script to interact with the corresponding test case within the applet. Similarly, on the card is defined an abstract class that defines three methods:

- a method `setUp()` to perform any memory allocation needed during the lifetime test case.
- a method `run()` used to launch the tests corresponding to the test case of interest, and
- a method `cleanUp()` used after the test is done to perform any clean-up.

The testing applet is capable of recognizing all the test cases and launching a particular test by executing its `run` method.

Our Eclipse environment integrates the Converter tool from Sun MicroSystems, which is used to convert a standard Java Card applet class into a JCA file during a first step. This file is completed pseudo-automatically by integrating the operations to be tested with the Java Card Assembly instructions, as we explain in the following paragraph. The second step consists in capgenerating the JCA file into a CAP file, so that the applet could be installed on any Java Card platform.

3.2 Modules

In this section, we describe the general benchmark framework that has been designed to achieve the MESURE objective. The methodology consists of different

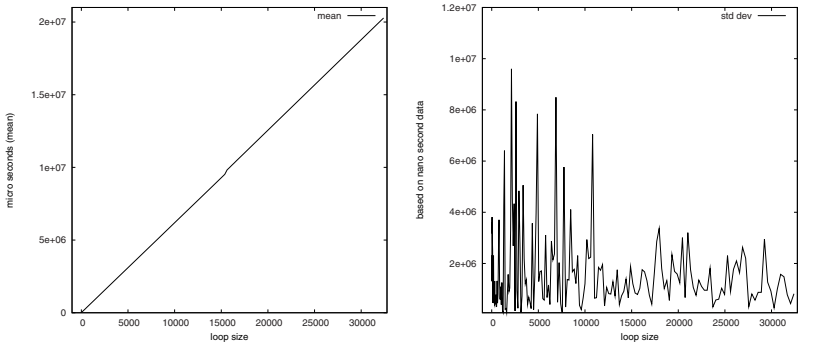

Fig. 2. Raw measurement and standard deviation

steps. The objective of the first step is to find the optimal parameters used to carry out correctly the tests. The tests cover the VM operations and the API methods. The obtained results are filtered by eliminating non-relevant measurements and values are isolated by drawing aside measurement noise. A profiler module is used to assign a mark to each benchmark type, hence allowing us to establish a performance index for each smart card profile used. In the following subsections, we detail every module composing the framework.

The bulk of the benchmark consists in performing time execution measurements while we send APDUs from the computer through the Card Acceptance Device (CAD) to the card. Each test (`run`) is performed a certain number of times (Y) to ensure reliability of the collected execution times , and withing each `run` method, we perform on the card a certain number of loops (L). L is coded on the byte P_2 of the APDUs which are sent to the on-card applications. The size of the loop performed on the card is $L = (P_2)^2$.

The Calibrate Module. The calibrate module computes the optimal parameters (such as the number of loops) needed to obtain measurements of a given precision.

Benchmarking the various different bytecodes and API entries takes time. At the same time, it is necessary to be precise enough when it comes to measuring those execution times. Furthermore, the end user of such a benchmark should be allowed to focus on a few key elements with a higher degree of precision. It is therefore necessary to devise a tool that let us decide what are the most appropriate parameters for the measurement.

Figure 2 depicts the evolution of the raw measurement, as well as its standard deviation, as we take 30 measurements for each available loop size of a test applet. As we can see, the measured execution time of an applet grows linearly with the number of loops being performed on the card (L). On the other hand, the perceived standard deviation on the different measurements varies randomly as the loop size increases, though with less and less peaks. Since a bigger loop size means a relatively more stable standard deviation, we can use both the

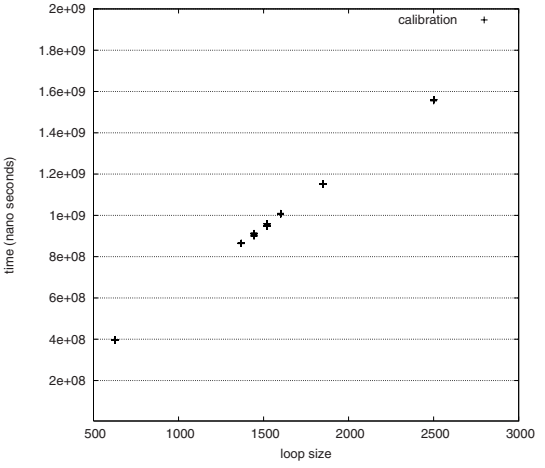

Fig. 3. A sample calibration

standard deviation and the mean measured execution time as a basis to assess the precision of the measurement as follows.

To assess the reliability of the measurements, we compare the value of the measurement with the standard deviation. The end user will need to specify this ratio between the average measurement and the standard deviation, as well as an optional mininum accepted value, which is set at one second by default.

With both the ratio and the minimal accepted value, as specified by the end user, we can test and try different values for the loop size to binary search and approach the ideal value. In figure 3, we try to calibrate a test by first trying out a loop size of 2500. The program decided that the set of 30 obtained values was too precise and therefore too time demanding. It then tried to evaluate the precision of the test for a loop size of 625. Since the measurements were below the minimum value, the program then tried to perform the same evaluation for a loop size of 1369, and so on, until we reached a loop size for which both conditions were satisfied.

The Bench Module. For a number of cycles, defined by the calibrate module, the bench module performs the measurements for:

- The VM byte codes
- The API methods
- The JCRE mechanisms (such as transactions)

The developement of some of the test applets is detailed in [18].

The Filter Module. Experimental errors lead to noise in the raw measurement experiments. This noise leads to imprecision in the measured values, making it difficult to interpret the results. In the smart card context, the noise is due to crossing the platform, the CAD and the terminal (measurement tools, OS, hardware).

The issues become: how to interpret the varying values and how to compare platforms when there is some noise in the results. The filter module uses a statistical design to extract meaningful information from noisy data. From multiple measurements for a given operation, the filter module uses the mean value μ of the set of measurements to guess the actual value, and the standard deviation σ of the measurements to quantify the spread of the measurements around the mean. Moreover, since the measurements respect the normal Gaussian distribution (see figure 4), a confidence interval $[\mu - (n \times \sigma), \mu + (n \times \sigma)]$, within which the confidence level is of $1 - a$, is used to help eliminate the measurements outside the confidence interval, where n and a are respectively the number of measurements and the temporal precision, and they are related by traditional statistical laws.

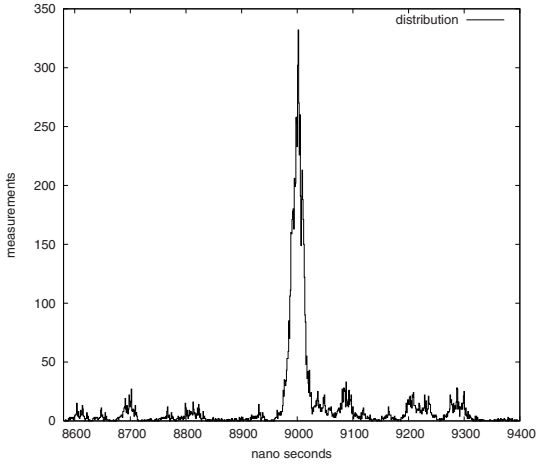

Fig. 4. The distribution of 10000 measured execution time

The Extractor Module. The extractor module is used to isolate the execution time of the features of interest among the mass of raw measurements that we gathered so far.

Benchmarking bytecodes and API methods within Java Card platforms requires some subtle means in order to obtain execution results that reflect as accurately as possible the actual isolated execution time of the feature of interest. This is because there exists a significant and non-predictable elapse of time between the beginning of the measure, characterized by the starting of the timer on the computer, and the actual execution of the bytecode of interest. This is also the case the other way around. Indeed, when performing a request on the card, the execution call has to travel several software and hardware layers down to the card's hardware and up to the card's VM (vice versa upon response). This non-predictability is mainly dependent on hardware characteristics of the benchmark environment (such as the card acceptance device (CAD), PC's hardware, etc), the OS level interferences, services and also on the PC's VM.

To minimize the effect of these interferences, we need to isolate the execution time of the features of interest, while ensuring that their execution time is sufficiently important to be measurable.

The maximization of the bytecodes execution time requires a test applet structure with a loop having a large upper bound, which will execute the bytecodes for a substantial amount of time. On the other hand, to achieve execution time isolation, we need to compute the isolated execution time of any auxiliary bytecode upon which the bytecode of interest is dependent. For example if `sadd` is the bytecode of interest, then the bytecodes that need to be executed prior to its execution are those in charge of loading its operands onto the stack, like two `sspush`. Thereafter we subtract the execution time of an empty loop and the execution time of the auxiliary bytecodes from that of the bytecode of interest to obtain the isolated execution time of the bytecode. As presented in figure 5, the actual test is performed within a method (`run`) to ensure that the stack is freed after each invocation, thus guaranteeing memory availability.

Applet framework	Test Case

`process() {`
`  i = 0`
`  While i <= L`
`  DO {`
`  run()`
`  i = i+1`
`  }`
`}`
	`run() {`
	`  op`$_0$
	`  op`$_1$
	`  ⋮`
	`  op`$_{n-1}$
	`  op`$_n$
	`}`

Fig. 5. Test framework for a bytecode op_0

In figure 5:

- L represents the chosen loop upper bound;
- op_n represents the operation of interest;
- op_i for $i \in [0..n-1]$ represents the auxiliary bytecodes necessary to perform the operation op_n.

To compute the mean isolated execution time of op_n we need to solve a system with the following equations:

$$\overline{M(op_n)} = \frac{\overline{m_L(op_n)} - \overline{m_L(Emptyloop)}}{L} - \sum_{i=0}^{n-1} \overline{M(op_i)}$$

Where:

- $\overline{M(op_i)}$ is the mean isolated execution time of the operation op_i.
- $m_L(op_i)$ is the mean global execution time of the operation op_i, including interferences coming from other operations performed during the measurement, both on the card and on the computer, with respect to a loop size L.

These other operations represent for example auxiliary bytecodes needed to execute the operation of interest, or OS and JVM specific operations. The mean is computed over a significant number of tests. It is the only value that is experimentally measured.
- *Emptyloop* represents the execution of a case where the `run` method does nothing.

The formula presented above implies that prior to computing $\overline{M(op_n)}$ we need to compute $\overline{M(op_i)}$ for $i \in [0..n-1]$. The system can be solved as long as the dependency relation between the operations is well founded, and that there is a set of operations that do not depend on any other operation.

The Profiler Module. In order to define performance references, our framework provides measurements that are specifically adapted to one of the following application domains:

- banking applications
- transport applications
- identity applications.

A Java Card VM is instrumented in order to count the different operations performed during the execution of a script for a given application. More precisely, this virtual machine is a simulated and proprietary VM executing on a workstation. This instrumentation method is rather simple to implement compared to a static analysis based methods, and can reach a good level of precision, but it requires a detailed knowledge of the applications and of the most significant scripts.

Some features related to bytecodes and API methods appeared to be necessary and the simulator was instrumented to give useful information such as:

- for the API methods:
  - the types and values of method parameters
  - the length of arrays passed as parameters,
- for the bytecodes:
  - the type and duration of arrays for array related bytecodes (load, astore, arraylength),
  - the transaction status when invoking the bytecode.

A simple utility tool has been developed to parse the log files generated by the instrumented Java Card VM, which builds a human-readable tree of method invocations and bytecode usage.

Thus, with the data obtained from the instrumented VM, we attribute for each application domain a number that represents the performance of some representative applets of the domain on the tested card. Each of these numbers is then used to compute a global performance mark.

We use weighted means for each domain dependent mark. Those weights are computed by monitoring how much each Java Card feature is used within a regular use of standard applets for a given domain. For instance, if we want to

test the card for a use in transport applications, we will use the statistics that we gathered with a set of representative transport applets to evaluate the impact of each feature of the card.

We are considering the measurement of the feature f on a card c for an application domain d. For a set of n_M extracted measurements $M^1_{c,f}, ..., M^{n_M}_{c,f}$ considered as significant for the feature f, we can determine a mean $\overline{M_{c,f}}$ modeling the performance of the platform for this feature.

Given n_C cards for which the feature f was measured, it is necessary to determine the reference mean execution time R_f, which will then serve as a basis of comparison for all subsequent tests.

Hence the "mark" $N_{c,f}$ of a card c for a feature f, is the relation between R_f and $\overline{M_{c,f}}$:

$$N_{c,f} = \frac{R_f}{\overline{M_{c,f}}}$$

However, this mark is not weighted. For each pair of a feature f and an application domain d, we associate a coefficient $\alpha_{f,d}$, which models the importance of f in d . The more a feature is used within typical applications of the domain, the bigger the coefficient:

$$\alpha_{f,d} = \frac{\beta_{f,d}}{\sum_{i=1}^{n_F} \beta_{i,d}}$$

where:

- $\beta_{f,d}$ is the total number of occurrences of the feature f in typical applications of the domain d.
- n_F is the total number of features involved in the test.

Therefore, the coefficient $\alpha_{f,d}$ represents the occurrence proportion of the feature of interest f among all the features.

Hence, given a feature f , a card c and a domain d , the "weighted mark" $W_{c,f,d}$ is computed as follows:

$$W_{c,f,d} = N_{c,f} \times \alpha_{f,d}$$

The "global mark" $P_{c,d}$ for a card c and for a domain d is then the sum of all weighted marks for the card. A general domain independant note for a card is computed as the mean of all the domain dependant marks.

3.3 Unused Features

The document [19] details the included and the excluded features. Only features related to the normal use phase of Java Card applications are considered here. Measuring the performance when installing, personalizing or deleting an application, is of less importance from the user's point of view. Moreover, these management operations are only performed once. As a consequence, the constructors of the Java Card API, as well as methods such as `Applet.register()`,

etc. are not measured. Besides, we focus on success paths and not on the failure ones, on account of their relevance. Then, failure cases such as the comparison methods of the Java Card API.equals(...) on a bad AID (`OwnerPIN.check(...)` on a bad PIN ...), as well as `Exception` classes are not taken into account. In the same respect, some bytecodes, that are never used in a regular application are not measured here.

4 State of the Art

Currently, there is no standard benchmark suite which can be used to demonstrate the use of the JCVM and to provide metrics for comparing Java Card platforms. In fact, even if numerous benchmarks have been developed surrounding the JVM (see 3), there are few works that attempt to evaluate the performances of smart cards. The first interesting initiative has been done by Jordi et al. in [17] where they study the performance of micro-payment for Java Card platforms, i.e., without PKI. Even if they consider Java Card platforms from distinct manufacturers, their tests are not complete as they involve mainly computing some hash functions on a given input, including the I/O operations. A more recent and complete work has been undertaken by Erdmann in [15]. This work mentions different application domains, and makes the distinction between I/O, cryptographic functions, JCRE and energy consumption. Infineon Technologies is the only provider of the tested cards for the different application domains. The software itself is not available. The work of Fischer in [16] compares the performance results given by a Java Card applet with the results of the equivalent native application. Another interesting work is that carried out by the IBM BlueZ secure systems group and concretized through a Master thesis [11]. JCOP framework has been used to perform a series of tests to cover the communication overhead, DES performance and reading and writing operations into the card's memory (RAM and EEPROM). Markantonakis in [9] presents some performance comparisons between the two most widely used terminal APIs, namely PC/SC and OCF. Papapanagiotou et al. in [10] evaluate the performance of two online certificate revocation and validation protocols on two different Java Card platforms in order to determine which protocol is more efficient for smart card use. Chaumette et al. in [13,14] show the performance of a Java Card grid with respect to the scalability of the grid and with different types of cards.

5 Conclusion

In this paper, we have proposed a methodology aiming at characterizing the performance of Java Card platforms by measuring different levels of benchmarks using measurement techniques to analyze the platform's performance. This work was undertaken as part of a project funded by the French administration MESURE. The Java Card Benchmarking framework is now accessible on-line (see [12]) since it is published as an open-source tool. Our work focuses on

measuring the excution time of the virtual machine bytecodes, the API methods and the JCRE mechanisms.

All the measured features are based on the Java Card 2.2 platforms. With the publication of the Java Card 3.0 specifications [20], two versions are proposed. While the Connected Edition features a new virtual web-oriented machine, the Classic Edition is based on an evolution of the Java Card Platform, Version 2.2.2 and targets more resource-constrained devices that support traditional applet-based applications. Hence, the majority of the features measured in Mesure tool will be reused in this edition. However, all the new features such as those based on 32-bit integers are not considered.

Currently, we are working on the prediction of the execution time of the applications, by using formal methods.

References

1. Cap, C.H., Maibaum, N., Heyden, L.: Extending the Data Storage Capabilities of a Java-based Smart card. In: Sixth IEEE Symposium on Computers and Communications (ISCC 2001). IEEE, Los Alamitos (2001)
2. Chen, Z.: Java Card Technology for Smart Cards: Architecture and Programmer's Guide. Addison Wesley, Reading (2000)
3. Douin, J.-M., Paradinas, P., Pradel, C.: Open Benchmark for Java Card Technology, e-Smart Conference (September 2004)
4. GemXpresso Reference Manual, Gemplus (1998)
5. Sm@rtCafe Reference Manual Giesecke & Devrient (1999)
6. Grimaud, G., Paradinas, P., Vétillard, E.: Measuring the performance of the Java Card Platform, Java One (May 2006)
7. Guyot, V., Boukhatem, N., Pujolle, G.: Smart Card performances to handle Session Mobility. ICI, IFIP/IEEE (September 2005)
8. Java Card 2.2.2 Specification (April 2006), `http://java.sun.com/products/javacard/`
9. Markantonakis, C.: Is the performance of smart card cryptographic functions the real bottleneck? In: 16th international conference on Information security: Trusted information: the new decade challenge, vol. 193, pp. 77–91. Kluwer, Dordrecht (2001)
10. Papapanagiotou, K., Markantonakis, C., Zhang, Q., Sirett, W.G., Mayes, K.: On the Performance of Certificate Revocation Protocols Based on a Java Card Certificate Client Implementation. In: 20th IFIP International Information Security Conference (Sec 2005) - Small Systems Security and Smart cards (May 2005)
11. Rehioui, K.: Java Card Performance Test Framework, Université de Nice, Sophia-Antipolis, IBM Research internship (September 2005)
12. The MESURE project MESURE, `http://mesure.gforge.inria.fr/Eng/Index`
13. Chaumette, S., Grange, P., Karray, A., Sauveron, D., Vignéras, P.: Secure distributed computing on a Java Card Grid, LaBRI, Université Bordeaux 1, 1331-04, (2004), `http://www.labri.fr/publications/paradis/2004/CGKSV04`
14. Atallah, E., Darrigade, F., Chaumette, S., Karray, A., Sauveron, D.: A Grid of Java Cards to Deal with Security Demanding Application Domains. In: 6th edition e-Smart conference & demos, Sophia Antipolis, Frensh Riviera (September 2005), `http://www.labri.fr/publications/paradis/2005/ADCKS05`

15. Erdmann, M.: Benchmarking von Java Card, LudwigMaximilians-Universität München, Institut für Informatik (May 2004)
16. Fischer, M.: Vergleich von Java und Native-Chipkarten Toolchains, Benchmarking, Messumgebung, LudwigMaximilians-Universität München, Institut für Informatik (2006)
17. Castellà-Roca, J., Domingo-Ferrer, J., Herrera-Joancomartí, J., Planes, J.: A Performance Comparison of Java Cards for Micropayment Implementation. In: CARDIS, pp. 19–38 (2000)
18. Paradinas, P., Cordry, J., Bouzefrane, S.: Performance Evaluation of Java Card Bytecodes WISTP, pp. 127–137 (May 2007)
19. Functionalities of the MESURE tools : `http://mesure.gforge.inria.fr/pub/documents/F2.1Functionalities1.0.pdf`
20. Java Card 3.0, `http://java.sun.com/javacard/3.0/`