Measurement Analysis when Benchmarking Java Card Platforms

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

[1] INRIA Rocquencourt 78150 Le Chesnay France
`Pierre.Paradinas@inria.fr`
[2] CNAM 292 rue Saint-Martin 75003 Paris France
`firstname.lastname@cnam.fr`

Abstract. The advent of the Java Card standard has been a major turning point in smart card technology. With the growing acceptance of this 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 level. This paper focuses on the validity of our time measurements on smart cards.

Key words: Java Card, Benchmark, Performance

1 Introduction

With more than 5 billion copies in 2008 [4], smart cards are an important device of todays information society. The development of the Java Card standard [1] made this device even more popular as it provides a secure, vendor-independent, ubiquitous Java platforms 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 (users, smart card manufacturers, card software providers, card users, card integrators, etc.). 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 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 [9] or IBM Research [16]). However, benchmarks are important in the smart card area because they contribute in discriminating companies products, especially when the products are standardised.

In this paper, we propose a general benchmarking solution through different steps that are essential for measuring the performance of the Java Card platforms. The emphasis here is towards the validation of the resulting tests in terms of accuracy and precision.

The remainder of this paper is organised as follows. In Section 2, we describe briefly some benchmarking attempts in the smart card area. In Section 3, an overview of the benchmarking framework is given. Section 4 analyses the obtained measurements using first a statistical approach, and then a precision reader before concluding the paper in Section 5.

2 Some attempts at measuring Java Card Performance

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 around the JVM, there are few works that attempt to evaluate the performance of smart cards.

The first interesting initiative has been done by Castellà et al. in [7] 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 [10]. 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 [11] compares the performance results given by a Java Card applet with the results of the equivalent native application.

Another interesting work has been carried out by the IBM BlueZ secure systems group and it was detailed in a Master thesis [16]. 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 memory (RAM and EEPROM).

Markantonakis in [13] presents some performance comparisons between the two most widely used terminal APIs, namely PC/SC and OCF.

Chaumette et al. in [5, 8] show the performance of a Java Card grid with respect to the scalability of the grid and with different types of cards.

3 General benchmarking framework

Our research work falls under the MESURE project [14], a project funded by the French administration (ANR), which aims at developing a set of open source tools to measure the performance of Java Card platforms. These benchmarking tools focus on Java Card 2.2 functionalities even if Java Card 3.0 specifications have been published since March 2008 [3], principally because until now there is no Java Card 3.0 platform in the market except some prototypes such as the one demonstrated by Gemalto during the Java One Conference in June 2008.

Moreover, since Java Card 3.0 proposes two editions: connected or web oriented edition and classic edition, our measuring tools can be reused to benchmark Java Card 3.0 classic edition platforms.

Our benchmarks have been developed under the Eclipse environment based on JDK 1.6, with JSR268 [2]. The underlying ISO 7816 smart card architecture forces us 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 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 equipments (the card reader and the workstation). Removing the noise added to a specific measurement is done with the computation of an average value extracted from multiple samples. As a consequence, each test is performed several times and some basic statistical calculations are used to filter the trustworthy results.

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.

As detailed in [6] the general benchmark framework follows different 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. Any measurement that is outside of a confidence interval can be considered as noisy. 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. 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 within each run method, we perform on the card a certain number of loops (L). L is coded on the byte `P2` of the APDUs which are sent to the on-card applications. The size of the loop performed on the card is $L = (\texttt{P2})^2$.

As [15] details, there is a way to isolate the time performance of fractions of code as small as a bytecode. As such we can isolate the performance of a simple `sadd` bytecode. We designed two tests (`run`). For each iteration of the L loops, each test calls a method. In the first of those tests (the reference test), the method called stacks up two numbers (`sspush`). In the second test (the operation test or, more precisely here, the `sadd` operation test), the method stacks up the two same numbers and performs a `sadd` (short addition). The difference of time performances between those two tests divided by L should give us the isolated time performance of a single `sadd`.

The `sadd` bytecode is a simple one, but there is primarily a need to validate our measurement method. Indeed, we need to know if our concept of measuring the performance of a smart card from the outside and isolating the performance of a single operation (bytecodes, and APIs entries) is valid.

4 Validation of the tests

4.1 Statistical correctness of the measurements

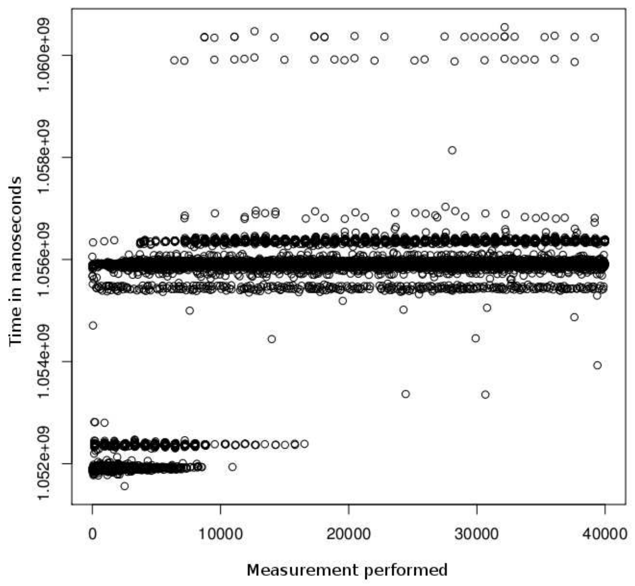
The expected distribution of any measurement is a normal distribution. According to Lilja [12], the arithmetic mean is an acceptable representative value for any given set of normally distributed time measurements (Lilja recommands at least 30 measurements). Nevertheless, Rehioui [16] pointed out that the results obtained via methods similar to ours were not normally distributed on IBM JCOP41 cards. Erdmann [10] cited similar problems with Infineon smart cards.

When measuring both the reference test and the operation test on several smart cards by different providers using different CADs (Cherry ST-1044U, FSC Smartcard-Reader USB 2A, GemPC Twin, Omnikey Cardman 2020, Omnikey Cardman 4040, Towitoko Chipdrive Micro, Xiring Teo), different host machines (with CPUs AMD Sempron 3100+, AMD X2 3800+, Intel Core2 Quad CPU Q9400), different OSs (Linux, Windows XP, Windows Vista), none of the time performances had a normal distribution (see figure 1 for a sample reference test performed on a card). The results were similar from one card to another in terms of distribution, even for different time values, and for different loop sizes. Changes in CAD, in host-side JVM, in task priority made no difference on the experimental distribution curve.

Testing the cards on Linux and on Windows XP or Windows Vista, on the other side, showed differences. Indeed, the recuring factor when measuring the performances with a terminal running Linux with PC/SC Lite and a CCID driver is the gap between peaks of distribution. The figure 1 shows the time values obtained for a set of performed measurements.

The peaks are often separated by 400ms and 100 ms steps which match some parts of the public code of PC/SC Lite and the CCID driver (see figure 2). With other CADs, the distribution shows similar steps with respect to the CAD driver source code. The peaks in the distribution from the measurements obtained on Windows are separated by 0.2 ms steps (see figure 4). Without having access to neither the source code of the PC/SC implementation on Windows nor the driver source codes, we can deduce that there must be some similarities in the source codes between the proprietary versions and the open source versions.

Fig. 1. Measurements of a reference test as the tests proceed under Linux, and the corresponding distribution curve $L = 41^2$



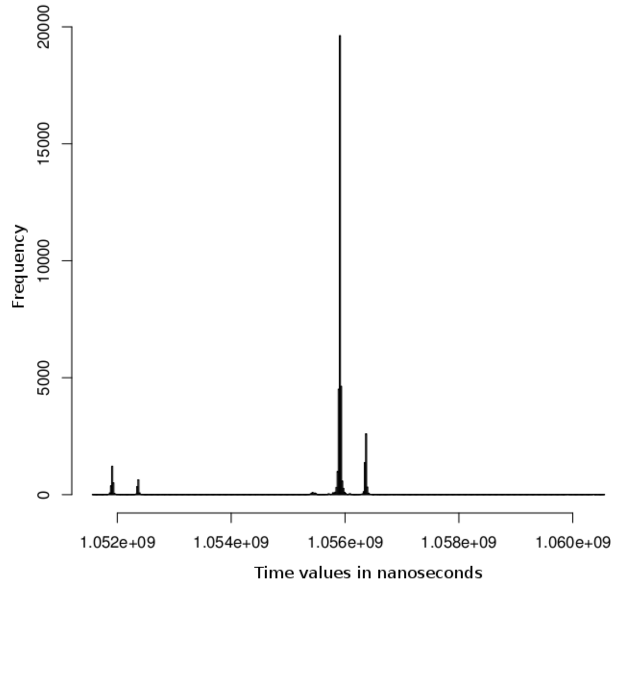


Fig. 2. Some lines from the PC/SC Lite and CCID driver source codes

```
pcscd.h:#define PCSCLITE_LOCK_POLL_RATE 100000
pcscd.h:#define PCSCLITE_STATUS_POLL_RATE 400000
winscard.c:SYS_USleep(PCSCLITE_LOCK_POLL_RATE);
winscard_clnt.c:SYS_USleep(PCSCLITE_STATUS_POLL_RATE + 10);
```

In order to check the normality of the results, we isolated some of the peaks of the distributions obtained with our measurements (see figure 3). The Shapiro-Wilk test is a well established statistical test used to verify the null hypothesis that a sample of data comes from a normally distributed population [17]. The result of such a test is a number $W \in [0, 1]$, with W close to 1 when the data is normally distributed. No set of value obtained by isolating a peak within a distribution gave us a satisfying W close to 1. For instance, considering the peak in figure 3, $W = 0.8442$, which is the highest value for W that we observed, with other values ranging as low as $W = 0.1384$. We conclude that the measurements we obtain, even if we consider a peak of distribution, are not normally distributed.

Rehioui [16] proposed an algorithm to locate the highest peak in a distribution to take the value of that peak as the correct measured value. However the algorithm does not literally try to locate the highest peak in the distribution curve, but, with each iteration of the algorithm, it removes the measurements that are to far away from the arithmetic mean of the measurements. The algorithm stops after several such iterations. We should be left with a certain percentage of the initial number of measurements. That particular percentage is determined by the user of the benchmark framework.

This algorithm is nevertheless futile when it comes to trying to determine the correct time performance value with a "comb" like distribution (see figure 4), or if the highest peak is relatively far from the mean (which would suggest that we have several other smaller peaks on the other side of the mean).

But what we are interested in is not exactly the raw measurement of the reference test and the operation test, but the differences between the operation measurements and the reference measurements.

Figure 5 shows two curves. The upper curve shows the time values obtained for a `sadd` operation test, while the lower curve shows the time values for the corresponding reference test. As we can see, each curve is subject to changes due to the non normal distribution of their respective measurements (that is, noises on the test platform). So it is difficult for us to choose the appropriate time value representing each curve. There is nevertheless a time difference between the two curves that is bigger than those variations. That is due to the execution of the supplementary `sadd` byte code in each iteration of the loop. For a sufficiently large loop size, the difference between those two curves is large enough so that it dwarfs the importance of those variations. Indeed, the supplementary bytecode

Fig. 3. Distribution of the measurement of a reference test : close up look at a peak in distribution $L = 41^2$

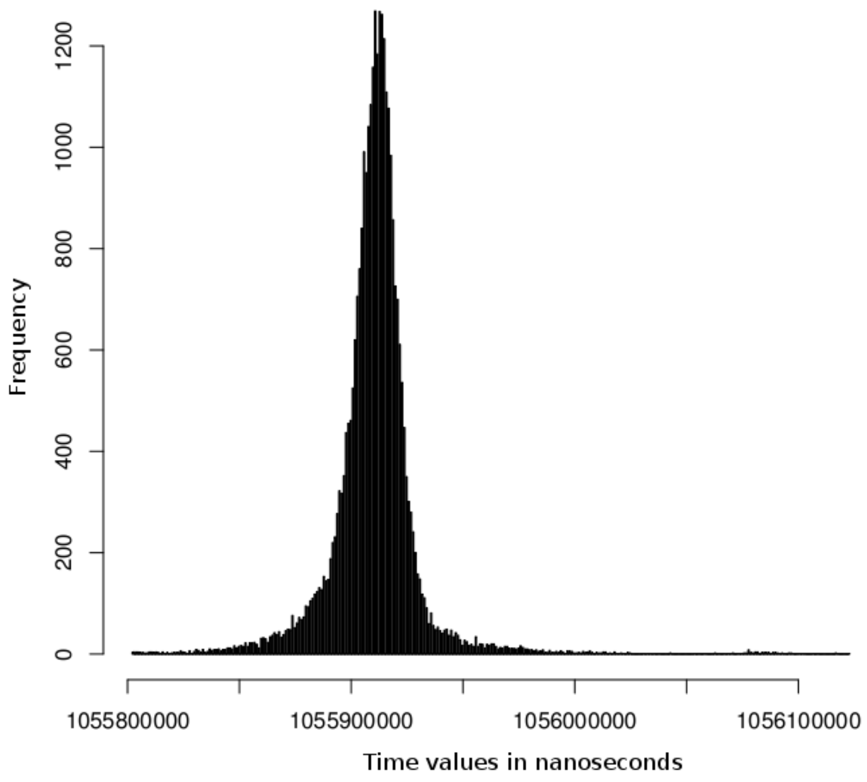
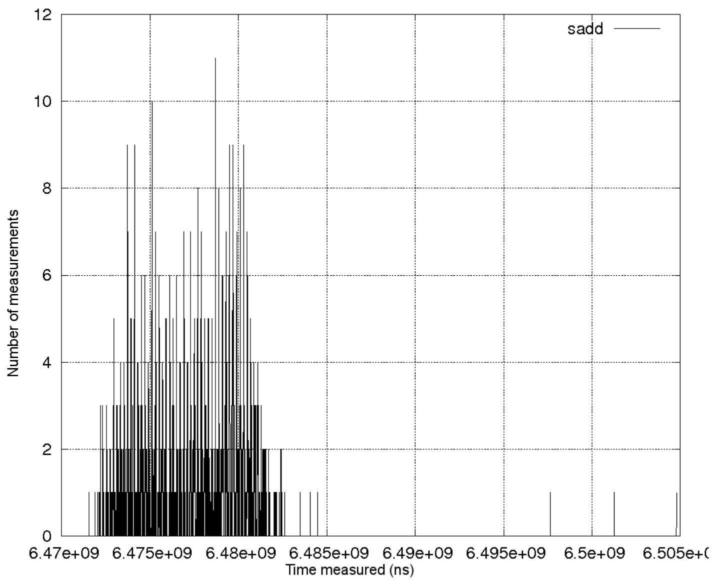


Fig. 4. Distribution of `sadd` operation measurements using Windows Vista, and a close up look at the distribution ($L = 90^2$)



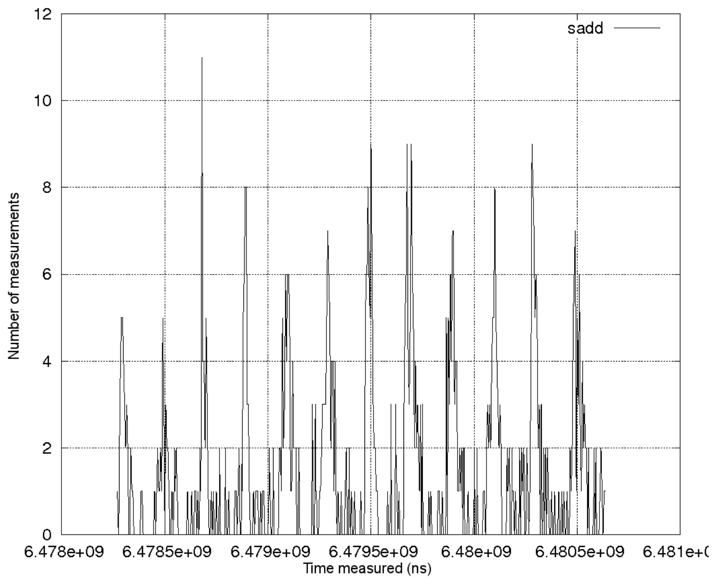
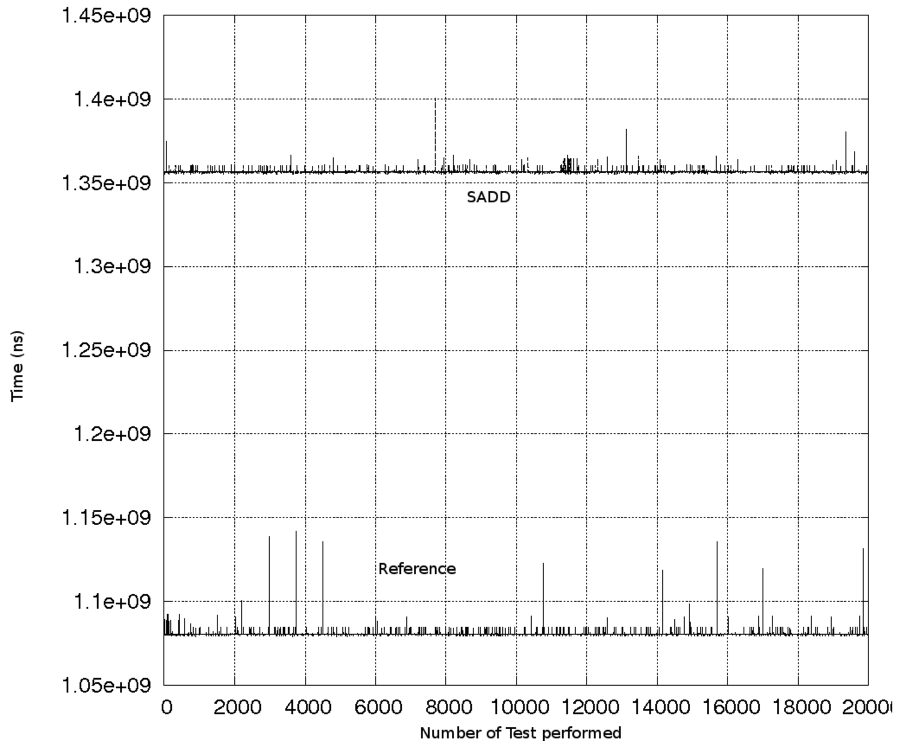


Fig. 5. Comparison between the `sadd` operation measurements and the corresponding reference measurements ($L = 41^2$)



is then performed a sufficiently large amount of times, so that it can have a large impact on the time performances.

So even though we don't have access to a set of normally distributed time values, for a given large loop size, the measurements could be accurate.

4.2 Validation through a precision CAD

We used a Micropross MP300 TC1 reader to verify the accuracy of our measurements. This is a smart card test platform, that is designed specifically to give accurate results, most particularly in terms of time analysis.

The results here are seemingly unaffected by noises on the host machine. With this test platform, we can precisely monitor the polarity changes on the contact of the smart card, that mark the I/Os.

We measured the time needed by a given smart card to reply to the same APDUs that we used with a regular CAD. We then tested the measured time values using the Shapiro-Wilk test, we observed $W \geq 0.96$, much closer to what we expected in the first place. So we can assume that the values are normally distributed for both the operation measurement and the reference measurement.

We subtracted each reference measurement value from each `sadd` operation measurement value, divided by the loop size to get a time values set that represents the time performance of an isolated `sadd` bytecode. Those new time values are normally distributed as well ($W = 0.9522$). On the resulting time value set, the arithmetic mean is 10611.57 ns and the standard deviation is 16.19524. According to [12], since we are dealing with a normal distribution, this arithmetic mean is an appropriate evaluation of the time needed to perform a `sadd` bytecode on this smart card.

Using a more traditional CAD (here, a Cardmann 4040, but we tried five different CADs) we performed 1000 measurements of the `sadd` operation test and 1000 measurements of the corresponding reference test. By subtracting each value obtained with the reference test from each of the values of the `sadd` operation test, and dividing by the loop size, we produced a new set of 1000000 time values. The new set of time values has an arithmetic mean of 10260.65 ns and a standard deviation of 52.46025.

The value we found with a regular CAD under Linux and without priority modification is just 3.42% away from the more accurate value found with the precision reader. Although this is a set of measurements that are not normally distributed ($W = 0.2432$), the arithmetic mean of our experimental noisy measurements seems to be a good approximation of the actual time it takes for this smart card to perform a `sadd`.

The same test under Windows Vista gave us a mean time of 11380.83 ns with a standard deviation of 100.7473, that is 7,24% away from the accurate value.

In conclusion, our data are noisy and faulty but despite a potentially very noisy test environment, our time measurements always provide a certain accuracy and a certain precision.

5 Conclusion

With the wide use of Java in smart card technology, there is a need to evaluate the performance and characteristics of these platforms in order to ascertain whether they fit the requirements of the different application domains. For the time being, there is no other open source benchmark solution for Java Card. The objective of our project [14] is to satisfy this need by providing a set of freely available tools, which, in the long term, will be used as a benchmark standard. In this paper, we have focused on the validation of our time isolation technique. Despite, the noise, our framework achieve some degree of accuracy and precision. Besides the portability of the benchmarking framework, this means that evaluating appropriately the performance of a smart card does not necessarily require a costly reader. Java Card 3.0 is a new step forward for this community. Our framework should still be relevant to the classic edition of this platform, but we have yet to test it.

References

1. Java Card 2.2.2 Specification, April 2006.
2. JSR 268: Java Smart Card I/O API, December 2006.
3. Java Card 3.0 Specification, March 2008.
4. Pierrick Arlot. Le marché de la carte à puce ne connait pas la crise. Technical report, Electronique international, 2008.
5. Eve Atallah, Franck Darrigade, Serge Chaumette, Achraf Karray, and Damien Sauveron. A grid of Java Cards to deal with security demanding application domains. In *6th edition e-Smart conference & demos*, September 2005. Sophia Antipolis, French Riviera.
6. Samia Bouzefrane, Julien Cordry, Hervé Meunier, and Pierre Paradinas. Evaluation of Java Card Performance. In *Eighth Smart Card Research and Advanced Application Conference CARDIS*, Egham, United Kingdom, September 2008.
7. Jordy Castellà-Roca, Josep Domingo-Ferrer, Jordi Herrera-Joancomati, and Jordi Planes. A performance comparison of Java Cards for micropayment implementation. In *CARDIS*, pages 19–38, 2000.
8. Serge Chaumette and Damien Sauveron. Some security problems raised by open multiapplication smart cards. In *10th Nordic Workshop on Secure IT-systems: NordSec 2005*, October 2005.
9. Jean-Michel Douin, Pierre Paradinas, and Cédric Pradel. Open Benchmark for Java Card Technology. In *e-Smart Conference*, September 2004.
10. Monika Erdmannn. Benchmarking von Java Cards. Master's thesis, Institut für Informatik der Ludwig-Maximilians-Universität München, 2004.
11. Mario Fischer. Vergleich von Java und native-chipkarten toolchains, benchmarking, messumgebung. Master's thesis, Institut für Informatik der Ludwig-Maximilians-Universität München, 2006.
12. David J. Lilja. *Measuring Computer Performance: A Practitioner's Guide*. Cambridge University Press, 2000.
13. Constantinos Markantonakis. Is the performance of smart card cryptographic functions the real bottleneck ? In *16th international conference on Information security: Trusted information: the new decade challenge*, volume 193, pages 77 – 91. Kluwer, 2001.

14. The MESURE project website. http://mesure.gforge.inria.fr.
15. Pierre Paradinas, Samia Bouzefrane, and Julien Cordry. Performance evaluation of java card bytecodes. In Springer, editor, *Workshop in Information Security Theory and Practices (WISTP)*, Heraklion, Greece, 2007.
16. Karima Rehioui. Java Card Performance Test Framework, September 2005. Université de Nice, Sophia-Antipolis, IBM Research internship.
17. S. S. Shapiro and M. B. Wilk. An analysis of variance test for normality (complete samples). *Biometrika, 52, 3 and 4*, pages 591–611, 1965.