Performance Evaluation of Java Card Bytecodes

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. The first experimental results show that bytecode execution time isolation is possible.

Keywords: Java Card, Benchmark, Performance.

1 Introduction

With more than one billion copies per year, smart cards are an important device of today's information society. The development of the Java Card standard made this device even more popular: capable of processing a subset of the platform independent, object oriented, and widely used programming language Java, the Java Card puts smart card technology at the disposal of many programmers and significantly shortens the time to market for smart card applications [3]. Moreover, cards are "open platforms" in the sense that programs (applets) can be added, that is, uploaded and executed on the platforms.

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 [7,8] or IBM Research [13]). However, benchmarks are important in the smart card area. Indeed, from smart card manufacturers point of view, standards will be more and more important in the smart card industry, as it is the case currently for the information technology domain. Furthermore, it is of primary importance to differentiate the products of the companies especially when the products are standardized. From a smart card customers point of view, benchmarks allow to understand the platform performance in terms of evaluation and prediction. It helps choose a service according to its QoS, its execution

D. Sauveron et al. (Eds.): WISTP 2007, LNCS 4462, pp. 127–137, 2007.

time or its memory consumption. Besides, being able to efficiently measure the performance of a cryptographic device such as a smart card in terms of time, memory or power consumption might be used to perform some security attacks and evaluations.

In this paper, we propose a general benchmarking solution to establish the execution time of Java Card bytecodes. We show that our proposed solution allows us to ascertain the feasibility of bytecode execution time isolation. Here, we restrict ourselves to presenting the comparative performances of arithmetic operations on several smart cards.

The remainder of this paper is organised as follows. In section 2, we give a brief introduction of the Java Card technology and present some related benchmarking solutions. Section 3 presents our benchmark framework and describes a general solution to achieve bytecode execution time isolation. We explain in section 4 how we revise the general solution to suit arithmetic operations benchmarking. Section 5 shows the results pertaining to arithmetic performance and also presents how the stability of the result is a function of the execution frequency of the bytecode under analysis. We present some future works in section 6 and conclude in section 7.

2 Java Card and Benchmarking

2.1 Java Card Technology

Java Card technology provides means of programming smart cards [6,1] 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 64KB, EEPROM memory (writable, persistent) between 16 and 32KB and RAM memory (writable, non-persistent) between 1 and 4KB. 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 initializes 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 case of Java powered smart cards, besides, the operating system, the card's ROM contains a Java Card Virtual Machine (JCVM) which implements a subset of the Java programming language and allows Java Card applets to run on the card.

A Java Card applet should implement the `install` method responsible for the initialization of the applet (usually it just calls the applet constructor) and a `process` method for handling incoming command APDUs and sending the response APDUs back to the host. There can be more than one applet existing on a single card, but there can be only one 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 Some Attempts for 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, thus allowing the Java Card users to take decision about which environments are most suitable for their needs. 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.

For example, SCCB (Smart Card CNAM Benchmark) from CEDRIC laboratory [7,8]), was initially a very ambitious project which aimed at measuring the performance of Java Card platforms. Unfortunately, the results obtained during experiments were not accurate because the measurements were initiated at the Java Card language level and were neglecting the basic operations defined at the bytecode level.

Another interesting work is that carried out by the IBM BlueZ secure systems group and concretized through a Master thesis [13]. 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 [10] presents some performance comparisons between the two most widely used terminal APIs, namely PC/SC and OCF. He measures some operations such as: connecting/disconnecting to the smart card reader, selecting the smart card application, sending APDUs, etc.

Guyot et al. in [9] describe how to handle session mobility by storing session information in smart card. In this special context, they evaluate the performance of smart cards by implementing real services and by observing how fast the cards could retrieve and suspend a given session.

Papapanagiotou et al. in [12] 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 [4,2] show the performance of a Java Card grid with respect to the scalability of the grid and with different types of cards.

Regarding the problem that we address here, the works of Guyot et al. and Papapanagiotou et al. are used in particular contexts and do not deal with Java Card platforms while Chaumette et al. deal with the performance of a grid rather than that of a single smart card.

3 General Benchmarking Framework

3.1 Introduction

Our research work falls under the MESURE project [11], 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. Currently, we have developed benchmarks covering some VM related characteristics, such as arithmetic. In this paper, we have chosen to present only the results pertaining to the arithmetic benchmarks because they are completely finished, compared to others (memory specific operations for example). 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 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 bytecodes. 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 to launch 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 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 Isolating Bytecode Execution Time

Benchmarking bytecodes within Java Card platforms requires some subtle means in order to obtain execution results that reflect as accurately as possible the actual execution time of the isolated execution time of an arithmetic bytecode. 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 round. 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 bytecodes 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 1, 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() {`
	`  op1`
	`  op2`
	`  .`
	`  .`
	`  opn`
	`  op0`
	`}`

Fig. 1. Test framework for a bytecode op_0

In figure 1 :

- L represents the chosen loop upper bound;
- op_0 represents the bytecode of interest;
- op_i for $i \in [1..n]$ represents the auxiliary bytecodes necessary to perform the bytecode op_0.

To compute the mean isolated execution time of op_0 we need to perform the following calculation:

$$\overline{M(op_0)} = \frac{\overline{m_L(op_0)} - \overline{m_L(Emptyloop)}}{L} - \sum_{i=1}^{n} \overline{M(op_i)}$$

Where :

- $\overline{M(op_i)}$ is the mean isolated execution time of the bytecode op_i.
- $m_L(op_i)$ is the mean global execution time of the bytecode 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 bytecode 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 $\overline{M(op_0)}$ we need to compute $\overline{M(op_i)}$ for $i \in [1..n]$.

4 Arithmetics

The benchmarking of arithmetic operations requires some fine-tunings. As arithmetic operations take in general a negligible amount of time to execute, the proposed general solution may not give satisfying results. More precisely, though having a large upper bound L ensures a certain degree of accuracy in our measurements, this involves making some very long and unpractical measurements. Whereas, if we perform our tests with a smaller upper bound, we can still have some cases where $\overline{m(op_i)} < \overline{m(Emptyloop)}$ (due to the small execution time of arithmetic operations), which are mainly due to sudden load changes within the benchmark platform. Consequently, this can affect adversely the mean execution time of the test. To minimize these undesirable situations, our solution, consists in executing repeatedly as many times as possible the bytecode of interest within the `run` method. Some smart cards might perform some security countermeasures (see [5]) that will degrade the execution time of multiple similar bytecodes executed in a row. In that case, our general solution will still work but we will need to perform a very large number of loops which will make the overall test tedious.

However, in the case of arithmetic operations, increasing the number of executions of the arithmetic bytecode by k times does not necessarily entail k executions of its auxiliary bytecodes ($op_1 ... op_n$) (generally `sspush` operations). This is due to the fact that an arithmetic operation always ends up pushing an operand onto the stack, corresponding to its result. Therefore, we can take advantage of this to optimize the overall benchmark execution time. When benchmarking an arithmetic operation requiring two operands, for instance a `sadd` operation, the `run` method will contain k executions of `sadd`, preceded by only $k+1$ executions of `sspush`, instead of $2k$. See figure 2.

The computation of the mean isolated execution time for a binary arithmetic operation op_0, when taking into account k executions of op_0 for every iteration, is presented as follows :

$$\overline{M(op_0)} = \frac{\overline{m_L(op_0)} - \overline{m_L(Emptyloop)}}{L \times k} - (k+1) \times \overline{M(\texttt{sspush num})}$$

Where :

$$\overline{M(\texttt{sspush num})} = \frac{\overline{m_L(\texttt{sspush num}))} - \overline{m_L(Emptyloop)}}{L \times k}$$

`run` method to isolate op_0	`run` method to isolate `sspush`

`run(){`
`sspush num`
`(k)` { `sspush num` / $\texttt{op}_0$
`}`
	`run(){`
	`sspush num`
	`(k) {sspush num`
	`}`

Fig. 2. `run` methods for binary arithmetic operations

5 Performance Results

5.1 Arithmetic Performance

We have evaluated the arithmetic performance of three Java Card 2.2 platforms denoted respectively 3060, 4045 and 2046. Cards 3060 and 2046 were designed respectively in 2006 and 2004 by the same manufacturer, whereas card 4045 was manufactured in 2004 by another provider. The benchmarks have been carried out by measuring the execution time of distinct arithmetic operations for each Java Card platform. The results presented in figure 3 show the isolated execution time of some arithmetic operations. As we can notice, the `sadd` bytecode for each card is approximately similar in time to the `ssub`, `sor`, `sand` and `sxor` bytecodes, which is normal since they are similar binary operations. We can also observe that `sneg` is the fastest one since it needs only one operand.

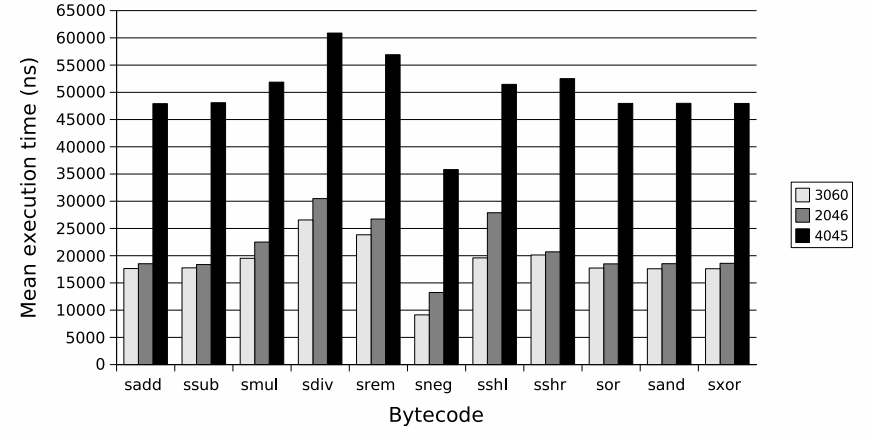

Fig. 3. Arithmetic performances of three cards

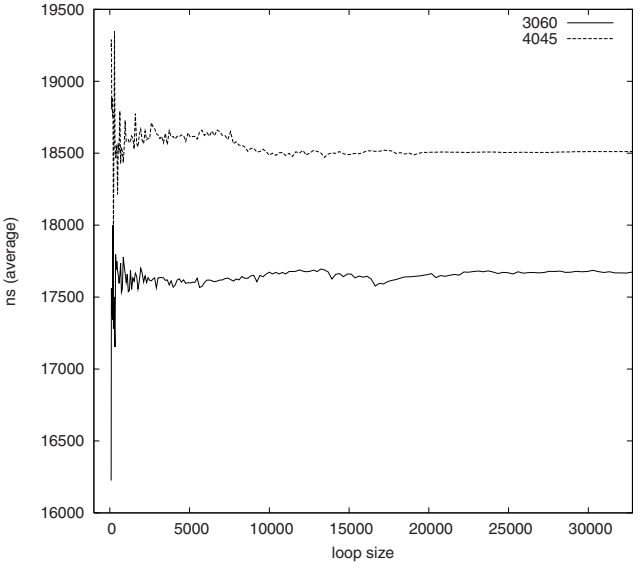

Fig. 4. Linearity evaluation on $\overline{M(\texttt{sadd})}$

With this test we have also been able to assess the characteristics of the three Java Card platforms vis-à-vis the arithmetic implementations of their JCVM. For instance, we can observe that for the cards 3060 and 4045, the `sshl` bytecode has nearly the same execution time as that of `smul` or `sshr`, whereas in card 2046 its execution time is nearer to that of `sdiv` or `srem`. From this observation, we can easily assume that `sshl` is implemented by a division in card 2046.

In conclusion, when comparing the performances of the three Java Card platforms, we can clearly see that card 3060 has slightly better performance than card 2046, whereas both of them outperform greatly card 4045.

5.2 Linearity of the Results

With our proposed bytecode execution time measurement solution, we expect that the mean execution time of the isolated bytecodes will stabilize after a certain loop size. We have checked for this linearity on the two cards 3060 and 4045. Figure 4 shows the mean execution time of a simple isolated `sadd` over 100 measures, based on each loop size. During this test, we have made use of the parameter `P2` of the APDU command to change the loop size.

As we can notice, the measures tend to reach a certain degree of stability as the loop size increases, though the results obtained for the two cards are dissimilar. We can also observe that their execution behaviour follows the same general pattern over the loop size range. This confirms the reliability of our proposed "bytecode execution time isolation" technique.

In general, the execution time stabilization is dependent upon factors such as the CAD, its driver, the computer OS, the CPU load during the test as well as

Instruction set					
Opcode	**Mnemonic**	**Reference loop**	**Opcode**	**Mnemonic**	**Reference loop**
0	nop	empty loop	117,118	<t>lookupswitch	1 <t>load | 1 goto
1	aconst_null	empty loop	119-121	<t>return	
2-8	sconst_<n>	empty loop	122	return	
9-15	iconst_<n>	empty loop	123-126	getstatic_<t>	empty loop
16-20	<t1><t2>push	empty loop	127-130	putstatic_<t>	1 <t>load
21	<t>load	empty loop	131-134	getfield_<t>	1 aload
24-35	<t>load_<n>	empty loop	135-138	putfield_<t>	1 aload | 1 <t>load
36-39	<t>aload	1 aload | 1 sload	139	invokevirtual	
40-42	<t>store	1 <t>load	140	invokespecial	
43-54	<t>store_<n>	1 <t>load	141	invokestatic	
55-58	<t>astore	1 aload | 1 sload | 1 <t>load	142	invokeinterface	
59	pop	1 sload	143	new	empty loop
60	pop2	2 sload	144	newarray	1 sload
61	dup	1 sload	145	anewarray	1 sload
62	dup2	2 sload	146	arraylength	1 aload
63	dup_x	1 bspush | n sload	147	athrow	
64	swap_x	1 bspush | n sload	148	checkcast	1 aload
65-88	<t><arithmetic_operation>	2 <t>load	149	instanceof	1 aload
89,90	<t>inc	empty loop	150,151	<t>inc_w	empty loop
91-94	<t1>2<t2>	1 <t1>load	152-159	if<cond>_w	1 sload
95	icmp	2 iload	160-167	if_<t>cmp<cond>_w	2 <t>load
96-103	if<cond>	1 sload	168	goto_w	nop
104-111	if_<t>cmp<cond>	2 <t>load	169-172	getfield_<t>_w	1 aload
112	goto	nop	173-176	getfield_<t>_this	empty loop
113	jsr		177-180	putfield_<t>_w	1 aload | 1 <t>load
114	ret		181-184	putfield_<t>_this	1 <t>load
115,116	<t>tableswitch	1 <t>load | 1 goto			

Fig. 5. Instruction Set

the card itself. For instance, in the case of the CAD, as the precision may vary from one CAD to another, the confidence in the results for a given loop size will vary. As a result, the loop size necessary to obtain an accurate and stable measure will depend generally upon the test environment, hence the needfulness for a loop size calibration prior to testing.

6 Future Works

6.1 Expanding the Test to Other Bytecodes

In the near future, we plan to expand the test to all bytecodes. Here also, our approach, at the outset, will be to track back any auxiliary bytecode necessary to satisfy the bytecode dependencies. The result of this dependency analysis is presented in figure 5. We categorise the terms upon which the bytecodes operate as follows:

$$\begin{aligned} t &::= a|b|i|s \\ n &::= m\_1|0|1|2|3|4|5 \\ cond &::= eq|ne|gt|ge|lt|le \end{aligned}$$

t represents the set of data types used, objects (a), bytes (b), integers (i) and shorts (s). n represents the allowed integer constants used. *cond* represents the different execution conditions.

For most of the cases, the test will follow the general framework presented in section 3. However, we will still be confronted to some exceptional cases such as those presented in the grayed background cells of the table. Indeed, the execution time of these bytecodes cannot be isolated. One possible solution is to pair bytecodes execution. For instance, we can measure the execution time of a method invocation (such as `invokestatic`) and the `return` bytecode as a whole.

6.2 Other Issues

In this paper we have focused on benchmarking Java Card bytecodes. But in the next few months, our objective will be to evaluate the execution time at the Java Card API level.

With the benchmark results, obtained at the bytecode and API levels, we will also be able to evaluate the performance of several execution scenarios of applets. The evaluation will require the analysis of an applet structure both at the bytecode and API levels to establish the bytecodes and methods used as well as their frequencies. The potential execution time of a given applet will be defined as a function of the frequency of methods/bytecodes and their benchmark results.

7 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 open source benchmark solution for Java Card. The objective of our project [11] 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 proposed, through our general benchmark framework, a "bytecode execution time isolation" technique that helps us assess the execution time of a bytecode, with OS level and hardware interferences removed.

We have shown via experimental tests that our technique produces accurate results with a confidence varying with respect to the test environment used. Indeed, stability of the result is strongly dependent on the frequency of the execution of the bytecode under scrutiny. We discussed that to obtain an accurate and stable measure, there is a need to calibrate the benchmark framework prior to testing.

References

1. Java card 2.2.2 specification, April 2006.
2. 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, Frensh Riviera.

3. Clemens H. Cap, Nico Maibaum, and Lars Heyden. Extending the data storage capabilities of a java-based smart card. In *Sixth IEEE Symposium on Computers and Communications (ISCC01)*. IEEE, 2001.
4. Serge Chaumette, Pascal Grange, Achraf Karray, Damien Sauveron, and Pierre Vignéras. Secure distributed computing on a java card grid. Technical Report 1331-04, LaBRI, Université Bordeaux 1, 2004.
5. 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.
6. Zhiqun Chen. *Java Card Technology for Smart Cards: Architecture and Programmer's Guide*. Addison Wesley, 2000.
7. Jean-Michel Douin, Pierre Paradinas, and Cédric Pradel. Open benchmark for java card technology. In *e-Smart Conference*, September 2004.
8. Gilles Grimaud, Pierre Paradinas, and Eric Vétillard. Measuring the performance of the java card platform. Java One, May 2006.
9. Vincent Guyot, Nadia Boukhatem, and Guy Pujolle. Smart card performances to handle session mobility. In *ICI*. IFIP/IEEE, September 2005.
10. 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.
11. The MESURE project website. http://cedric.cnam.fr/MESURE.
12. Konstantinos Papapanagiotou, Constantinos Markantonakis, Qing Zhang, William G. Sirett, and Keith Mayes. 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.
13. Karima Rehioui. Java card performance test framework, September 2005. Université de Nice, Sophia-Antipolis, IBM Research internship.