Journal of Universal Computer Science, vol. 16, no. 4 (2010), 519-530
submitted: 10/10/09, accepted: 21/12/09, appeared: 28/2/10 © J.UCS

Performance Optimizations for DAA Signatures on Java enabled Platforms

Kurt Dietrich
(Institute for Applied Information Processing and Communications (IAIK)
Graz University of Technology
Inffeldgasse 16a, 8010 Graz, Austria
Kurt.Dietrich@iaik.tugraz.at)

Franz Röck
(Institute for Applied Information Processing and Communications (IAIK)
Graz University of Technology
Inffeldgasse 16a, 8010 Graz, Austria
Franz.Roeck@student.tugraz.at)

Abstract: With the spreading of embedded and mobile devices, public-key cryptography has become an important feature for securing communication and protecting personal data. However, the computational requirements of public-key cryptosystems are often beyond the constraints embedded processors are bound to. This is especially true for cryptosystems that make heavy use of modular exponentiation like the Direct Anonymous Attestation scheme. The most popular algorithm for modular exponentiation is the Montgomery exponentiation based on sliding window technology. This technology offers several configuration options in order to get the best trade-off between the amount of precomputations and multiplications that are required for different exponentiation operands. Consequently, the optimum configuration and best parameters for receiving the highest performance gain are of interest. In this paper, we analyse different approaches for improving the performance of modular exponentiations with respect to the DAA scheme on Java enabled platforms. In particular, we analyse the optimal parameter setting for the Montgomery exponentiation and investigate how natively executed modular multiplications and modular reductions, with respect to a minimum of native code involved, can be integrated to improve the performance of mobile Java applications. Our experimental results show that the optimal setup of the Montgomery algorithm for a single modular exponentiation differs from the optimal setup used for the combination of all operations and operands used in the Direct Anonymous Attestation scheme. We also show that it is possible to get an immense performance gain by executing small parts of critical arithmetic operations natively on the platform thereby, not reducing the flexibility of mobile Java code.

Key Words: Trusted Computing, DAA, anonymous credentials, remote attestation
Category: L.4, K.6.5

1 Introduction

Today, nearly every mobile phone is equipped with a *Java virtual machine* [Yellin, Lindholm (1999)] which allows the device owner to install and run different applications *over-the-air* [Ortiz (2002)]. However, Java bytecode has the drawback to run slower than optimized C or Assembler code. As many important algorithms used for public-key cryptography such as RSA or the Direct Anonymous Attestation [Brickell, Camenisch, Chen (2004)] scheme rely on computation-intensive arithmetic operations such as modular exponentiations, this drawback has enormous effects on the execution speed. The modular exponentiations required for these operations involve very long integers, typically ranging from 512 to 2048 bits. A modular exponentiation is generally realized through a sequence of modular multiplications and consumes the majority of the execution time in inner loops. Improving the performance of these loop operations, therefore, has a significant impact on the total execution time of public-key cryptosystems.

The most prominent representative of a public-key cryptosystem used in Trusted Computing is the DAA scheme. This scheme involves many modular exponentiations, for example, creating a DAA signature requires the host to compute:

$$T_{host} = T_{tpm} * (A * S^{w})^{r_1} * S^{r_{\bar{\nu}}} \mod N \tag{1}$$

which involves three modular exponentiations and three multiplication with bases typically ranging up to 2048 bits and exponents ranging from 344 to 2737 bits [Brickell et al. (2005)]. In addition, the TPM has to compute:

$$T_{tpm} = R_0^{r_{f_0}} * R_1^{r_{f_1}} * S^{r_\nu} \mod N \tag{2}$$

while on general Trusted Computing (TC) enabled platforms, these computations (2) are executed inside the TPM, in case of virtual TPMs or mobile trusted modules (MTMs) [TCG-Mobile-Phone-Working-Group (2007)], [Dietrich and Winter (2009)] these exponentiations are done solely in software and are, therefore, performed on the platforms main CPU. According to (1) and (2), the host has to compute six modular exponentiations in total. Consequently, the most efficient implementation and parameter setting of the long integer modular arithmetic is crucial for the overall computation speed.

The performance of these computations can be improved in different ways. In this article, we discuss two of them: the first method addresses the used exponentiation algorithm. Many of these algorithms use windowing techniques that involve precomputing values that are later required for the actual multiplications. Therefore, it is essential to find the optimal distribution between the number of precomputated values and the number of modular multiplications in order to get the maximum overall performance. This distribution depends on the used window size - a larger window means more precomputed values and lesser multiplications. However, the optimal window size depends on the operands and their lengths used for the modular exponention and, therefore, varies for different cryptosystems. To be more exact, the window size varies for different modular exponentiations, depending on their operands, which is also the case when creating DAA signatures.

A discussion of our results in finding the optimal parameters for the DAA scheme is given in Section 2.4.

The second method addressed focuses on the Java architecture: Java offers the option of Just-in-Time (JIT) compilation [Krall (1998)] or *Dynamic-Ahead-Compilation*

(DAC). However, these options are not available on many embedded virtual machines [JSR 139 Process (2004)]. Moreover, specialized operations like modular arithmetics have a significant performance advantage when implemented specific for a certain platform instead of compiled by a JIT compiler. In order to take advantage of these performance gains with respect to the portability of Java applications, only the crucial parts of the arithmetic computations are moved to the native platform. These operations include the modular multiplication, modular reduction and squaring operations. The required modifications to the algorithm implementations and the different options for platform access from Java are discussed in Section 3. The remainder of this article is organised as follows: in Section 2, a brief introduction into the modular arithmetic that is used for our experiments is given. Finally, Section 4 includes a summary and a discussion of further analysis.

2 Long Integer Arithmetic

Many important public-key cryptosystems, such as RSA or Diffie-Hellman, rely on modular exponentiation, for example, operations of the form $c = m^e \mod n$ where m, e and n are long integer values [Menezes et al. (c1997)]. Although a variety of algorithms for modular exponentiation in literature exist, all of them can be reduced to modular multiplications, reductions and squarings. Hence, reducing the number of multiplications and reductions is desirable. The modular multiplication can be achieved in different ways. One method for achieving efficient reduction is discussed in the following Sections.

2.1 Sliding Window Exponentiation

Many exponentiation algorithms use a sliding window technique as discussed by Kaya Koc [Kaya Koc (1995)].

This technique reduces the average number of multiplications required for exponentiation. It is based on repeated squarings and multiplications but also includes a precomputation step. The number of these precomputations depends on the factor *k*. This factor denotes the size of our *window* - the larger the size of *k* the more precomputation have to be done and the possibility of fewer multiplications excluding precomputations is given. However, precomputing values is time consuming and might exceed the performance gain achieved by reducing the number of multiplications. Consequently, experimenting with *k* in order to find the optimal value for certain exponent lengths is recommended.

We have chosen an algorithm with a fixed window size as this kind of algorithms is one of the most widely used. Moreover, it is one of the fasted algorithm for modular exponentiation, it is easy to implement and provides a good trade-off between codesize and performance [Kaya Koc and Acar (1996)]. A comparison with algorithms that use variable window sizes is out of scope of this paper.

Algorithm 1 Sliding Window Exponentiation

```
Require: $g, e = (e_t e_{t-1} ... e_1 e_0)_2$ with $e_t = 1$, and an integer $k \geq 1$.
Ensure: $g^e$
 1: Precomputation
 2: $g_1 \Leftarrow g, g_2 \Leftarrow g^2$
 3: for $i = 1$ to $(2^{k-1} - 1)$ do
 4:     $g_{2i+1} \Leftarrow g_{2i-1} * g_2$
 5: end for
 6: end Precomputation
 7: $A \Leftarrow 1, i \Leftarrow t$
 8: while $i \geq 0$ do
 9:     if $e_i = 0$ then
10:         $A \Leftarrow A^2$,$i \Leftarrow i - 1$
11:     else
12:         find the longest bitstring $e_i e_{i-1} ... e_l$ such that $i - l + 1 \leq k$ and $e_l = 1$, and
            do the following:
13:         $A \Leftarrow A^{2^{i-l+1}} * g_{(e_i e_{i-1} ... e_l)_2}, i \Leftarrow l - 1$
14:     end if
15: end while
16: Return $(A)$
```

2.2 Montgomery Reduction

The Montgomery Reduction is a very efficient algorithm used for reduction in large modular multiplications. As shown in Algorithm 2, the reduction is done by calculating TR^{-1}. By choosing the so called *Montgomery residual factor* $R = b^n = 2^n$, this can be done very efficiently. Before doing modular multiplications, the factors have to be initialized by multiplying them with R, so that in the end, after multiplying the final result with R^{-1}, the *Montgomery residual factor* cancels out. The Montgomery Reduction also prohibits the intermediate values arising from multiplications and squarings to be large.

2.3 Efficient Squaring

For the squaring operations, we have chosen the *separated operand scanning method* as recommended in [Kaya Koc (1995)]. Although building the square of a long integer has an performance advantage over multiplying the value by itself, it can never be more than twice as fast as a multiplication [Menezes et al. (c1997)].

Algorithm 2 Montgomery reduction

```
Input: integers $m = (m_{n-1}...m_1m_0)_b$ with $gcd(m,b) = 1$, $R = b^n$,$m' = -m^{-1} \mod b$, and $T = (t_{2n-1}...t_1t_0)_b < mR$.
Output: $TR^{-1} \mod m$
 1: $A \Leftarrow T$ (Notation: $A = (a_{2n-1}...a_1a_0)_b$)
 2: for $i = 0$ to $(n-1)$ do
 3:     $u_i \Leftarrow a_i m' \mod b$
 4:     $A \Leftarrow A + u_i m b^i$
 5: end for
 6: $A \Leftarrow A/b^n$
 7: if $A \geq m$ then
 8:     $A \Leftarrow A - m$
 9: end if
10: $Return(A)$
```

Algorithm 3 Multiple-precision squaring

```
Input: positive integer $x = (x_{t-1}x_{t-2}...x_1x_0)_b$
Output: $x * x = x2$ in radix $b$ representation
 1: for $i = 0$ to $(2t-1)$ do
 2:     $w_i \Leftarrow 0$
 3: end for
 4: for $i = 0$ to $(t-1)$ do
 5:     $(uv)_b \Leftarrow w_{2i} + x_i * x_i$, $w_{2i} \Leftarrow v$, $c \Leftarrow u$
 6:     for $j = (i+1)$ to $(t-1)$ do
 7:         $(uv)_b \Leftarrow w_{i+j} + 2x_j * x_i + c$, $w_{i+j} \Leftarrow v$, $c \Leftarrow u$
 8:     end for
 9:     $w_{i+t} \Leftarrow u$
10: end for
11: $Return((w_{2t-1}w_{2t-2}...w_1w_0)_b)$
```

2.4 Results

In this section, an analysis of the performance results of the exponentiation process is given. A first investigation of a single modular exponentiation revealed the average time for performing one single modular multiplication and one single squaring operation, as shown in Table 1. The results stem from the observation of the modular exponentiation with varying exponents of the coefficient $S^{r_{\bar{v}}}$: This exponentiation which requires an exponent ($r_{\bar{v}}$) of size 2737 bits and a base (S) of a size up to 2048 bits is therefore one of the largest coefficients in Formula (1). The parameters where chosen according to the proposed values in the DAA scheme [Brickell et al. (2005)].

The values in Table 1 represent the average results when using a window size of

	multiply	**square**
operations	389	2737
meantime PC	0.17 ms	0.12 ms
meantime ARM9	19 ms	15 ms

Table 1: Performance of a single arithmetic operation

6. By varying the window size, the average performance values change according to Figure 1. The best performance is given at a windowsize of 7 that are $2^7 = 128$ precomputations which requires a total time of 625 ms to complete on the ARM9 platform and about 8.02 ms on the PC platform. (Note that all performance values were created with a modulus length of 2048 bits.)

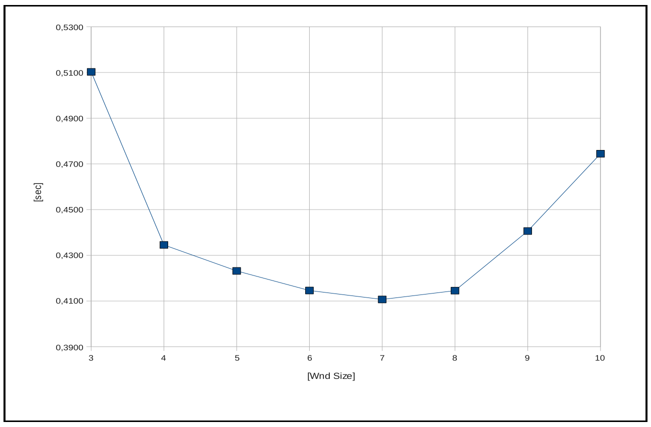

Figure 1: Performance values for different windowsizes - single exponentiation

Combining the precomputation, the modular multiplications, reductions and squarings together, we get total time required for one single modular exponention as shown in Table 2.

	PC	**ARM9**
modExp	0.41 s	50.14 s

Table 2: Performance of a single modular exponentiation

However, the measured values are only valid for this specific modular exponentiation with the given parameters. A sequence of modular exponentiations as used in DAA for creating a signature (1) (2) that involves operands with different base lengths and different exponent lengths show a different behavior and therefore require, a different average window size.

Figure 2 shows the computation speed in relation to the window size for a complete DAA signature. For creating a DAA signature, an optimum average windowsize of 6 is the best choice for the modular exponentiations. This setting results in 64 precomputated values per modular exponentiation.

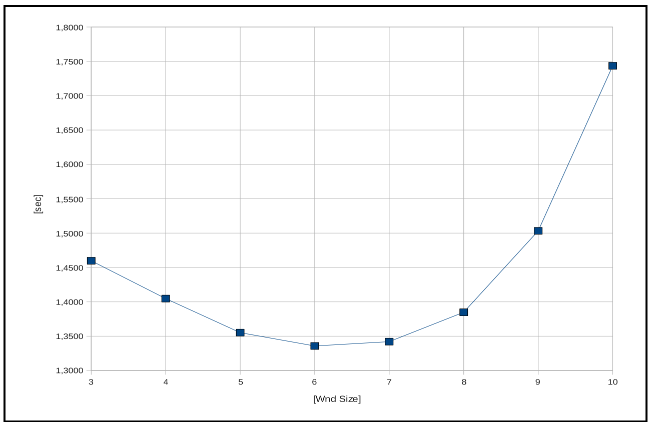

Figure 2: Performance values for different windowsizes - DAA sign

Keep in mind that a complete DAA signature also includes the computation of $s_n = r_n + c * X$ for the private parameters X. These private parameters include the private keys f_0, f_1, ν and credential parameters $e, \bar{\nu}$ as well as the computation of the hash $c = H(H(H(n\|R_0\|R_1\|S\|Z\|\|T\|\tilde{T}\|n_\nu)\|n_t)\|m)$ of the system and signature parameters as well as the the message *m*. Nevertheless, our investigation revealed that although these computations have an influence on the overall computation time, they are neglible when selecting the optimal window size for the exponentiations.

2.4.1 Testequipment

For our experiments, we used an ARM9 based micro processor platform as it is used in commom mobile phones and a standard PC with a 2.00 GHz Semptorm CPU. The Java Virtual machines used for our investigations were a SUN Java 1.6 on the PC and a proprietary VM from SonyEricsson.

3 Java Native Calls

A major performance improvement can be achieved by executing the critical operations natively on the platform. Java applications are running in a sand-box and, therefore, can not execute native code a priori. However, different methods to overcome this constraint exist. In order to access platform specific functions and equipment, Java offers an interface, the Java native interface (JNI) [SUN Microsystems (1997)] that allows to leave the virtual machine's sandbox, execute native code and return to original execution path of the Java application. Moreover, the interface provides a mechanism to transfer data to and from the Java application to the native code for processing.

The calls to the native world are associated with a small amount of time overhead for the native function call itself and the amount of data being transferred. Finding out how large the costs for native calls exactly are, is rather difficult, as it depends on the implementation of the JNI in the underlaying virtual machine. A native function call itself is estimated to be two to three times slower than a pure java function call on a typical virtual machine [Kesselman (2000)]. Moreover, the transfer overhead depends on how the transfer is done, as various possibilities how to achieve that exist. A good advice is to avoid memcopies. Instead of doing a complete copy of the parameters from the Java heap to the C heap, the native function can also get direct access to the data inside the Java object heap, which results in an substantial performance-boost compared to a native call that uses memcopy for data transfer [Kesselman (2000)] (Chapter 9.2.6). Because of these costs, it is reasonable to reduce the number of native calls as this overhead could lead to a performance loss. However, in case of complex arithmetic operations, the time required for the computations outweighs the time required for data transfer.

Another mechanism to allow Java applications execute native code is to integrate the code directly into the virtual machine [Sun Microsystems (2002)] where the VM offers an interface to the native code for applications executed by itself. This technique is used on many mobile JVMs as it is more lightweight than the JNI framework and easily portable to different platforms. However, the integration of the native code has to be provided by the VM manufacturer or the platform vendor.

As discussed in Section 2, modular exponentiations can be reduced to modular multiplications, reductions and squarings. These operations are the most time consuming ones when taking the whole exponentiation computation into account. Speeding up these computations results in a remarkable improvement of the whole operation. In our approach, we move these computations to the platform's execution environment. In detail, we now perform multiplications, squarings and reductions on the native platform. Although the overall number of these operations executed, of course, stays the same as when executed in pure Java, the performance gain is tremendous. Table 4 gives an overview of the achieved results.

The speed of a single multiplication or a single squaring executed in native code is about 3 times faster compared to the same code executed in Java on the PC platform. On the ARM9 platform, the performance gain is about 5 times. Note that all multiplications and squarings already include the Montgomery reduction step. Having in mind that Java on the PC platform uses JIT compilation, the arithmetic operations done in native code

	Java ARM9	**native ARM9**	**JIT Java PC**	**native PC**
mult.	19 ms	4 ms	0.165 ms	0.057 ms
squ.	15 ms	3 ms	0.128 ms	0.044 ms

Table 3: Average performance values of a single modular exponentiation with windowsize 6

even outrun the native code generated by the JIT compiler.

	Java ARM9	**native ARM9**	**JIT Java PC**	**native PC**
Sign	162.63 s	40.36 s	1.34 s	0.47 s
Verify	80.40 s	21.07 s	0.66 s	0.24 s

Table 4: DAA Signature Times

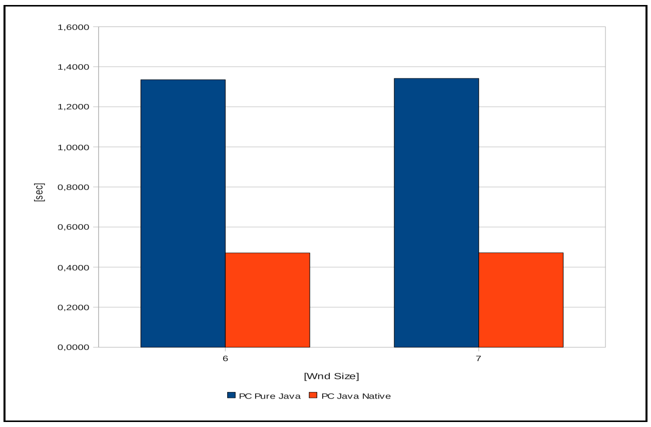

Figure 3: Pure Java and Java native code performance difference of DAA-sign (PC)

3.1 Deployment of the Native Library

The Java virtualization model allows mobile applications to be downloaded and executed more or less independent of the operating system and platform configuration. However, the involvement of specific platform code violates this assumption and a method for deploying our native code optimizations is required.

Using the JNI interface, the platform dependent code can be distributed by a simple mechanism. Java applications are typically deployed by packing the compiled Java classfiles together in a single archive file, which is then delivered to the target platform.

This archive file can also contain the native code library. When starting the application, the library can be extracted and copied to the native filesystem on the target platform where the JVM is searching for dynamic libraries. After the library has been stored on the platform, the application gives a command to the JVM to load the library. For the

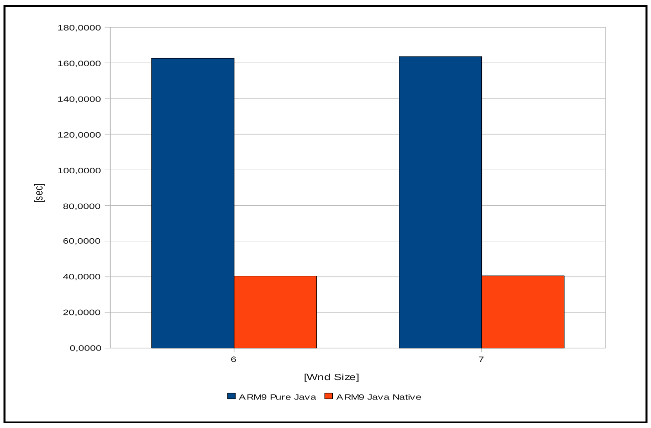

Figure 4: Pure Java and Java native code performance difference of DAA-sign (ARM9)

k-native interface, the situation is different. As the native code is already on the platform - to be more exact, it resides inside the JVM, the application is not required to carry the native code by itself. The drawback here is that this option is only available on specific JVMs. The native optimisations have to be integrated either by the platform integrator or the JVM vendor. In this case, it is reasonable that the application also carries the required arithmetic implementation in pure java and checks the presence of the native speedups before execution. If they are not present, the application has to use the arithmetic implemented in pure Java.

4 Conclusion

In this paper, we analysed two different approaches for optimising and increasing the performance of the DAA scheme on Java enabled platforms. The first approach addressed improvements of the modular exponentiation by finding the optimal window size for sliding window exponentiation algorithms with a fixed windows size.

We have shown that the optimal window size, for the used sliding window technology, is six for the combination of all modular exponentiations in the DAA signature and seven for a single modular exponentiation. Although it would be possible to estimate the optimal window size for each single modular exponentiation of (1), the management-overhead would increase because one would have to keep track of which exponentiation operation is currently executed. Consequently, we suggest to use the obtained mean-value.

The second approach addressed special features provided by Java. We have shown that moving the critical operations of a modular exponentiation i.e. modular multiplications, squarings and reductions to native functions, results in a performance-gain thereby providing a good balance between native code and pure Java code.

With a factor of three on the PC and a factor of four on an ARM9 processor, we could achieve a major performance improvement. However, the ARM9 is relatively slow compared to the PC, so, the factor of four brings us from 80 seconds for a verification down to approximately 20 seconds on this platform, which is already close for a possible practical use.

Unfortunately, this benefit is limited to platforms and JVMs with native support. The 40 seconds, which are required for a signature on an ARM9 with native functions, are still far away from practical use although they are already much better than the 160 seconds required for a signature computed in pure Java. The advantages of the native functions can not only be used for our DAA-signature, but all cryptography algorithms based on large number exponentiation should be able to reach a remarkable performance-gain when using native functions for their math operations.

Performance improvements could also be achieved by using dedicated security hardware for the computations which is typically much faster than software implementations. There are different approaches to investigate because the security hardware could be the SIM card that is attached to every mobile phone, it could be specific instruction set extensions [Grobschadl et al. (2007)] that are integrated in the main CPU or it could be Secure elements either fix attached to the device or removeable as part of an SD card.

Another idea could be to change the cryptographic primitives of the DAA scheme towards pairing-based cryptography as suggested in [Galbraith and Paterson (2008)].

Consequently, further research in the area of performance improvements for DAA should focus on the integration of dedicated cryptographic micro controller and improved cryptographic algorithms

References

[Brickell, Camenisch, Chen (2004)] Brickell, E., Camenisch, J., Chen, L.: "Direct Anonymous Attestation"; In Proceedings of 11th ACM Conference on Computer and Communications Security, ACM Press, 2004.

[Brickell et al. (2005)] Chris Mitchell: "Trusted Computing (Professional Applications of Computing)"; IEEE Press; Piscataway, NJ, USA; p. 143-174; ISBN-0863415253 (2005).

[Kaya Koc (1995)] Çetin Kaya Koc: "Analysis of sliding window techniques for exponentiation"; Computers and Mathematics with Applications; 30 (1995), 17–24.

[Kaya Koc and Acar (1996)] Çetin Kaya Koc, Acar, T.: "Analyzing and comparing montgomery multiplication algorithms"; IEEE Micro; 16 (1996), 26–33.

[Dietrich and Winter (2009)] Dietrich, K., Winter, J.: "Implementation aspects of mobile and embedded trusted computing."; L. Chen, C. J. Mitchell, A. Martin, eds., TRUST; volume 5471 of Lecture Notes in Computer Science; 29–44; Springer, 2009.

[Galbraith and Paterson (2008)] Galbraith, S. D., Paterson, K. G., eds.: Pairing-Based Cryptography - Pairing 2008, Second International Conference, Egham, UK, September 1-3, 2008. Proceedings; volume 5209 of Lecture Notes in Computer Science; Springer, 2008.

[Grobschadl et al. (2007)] Grobschadl, J., Tillich, S., Szekely, A.: "Performance evaluation of instruction set extensions for long integer modular arithmetic on a sparc v8 processor"; Digital Systems Design, Euromicro Symposium on; 0 (2007), 680–689.

[Kesselman (2000)] Kesselman, S. J.: Java Platform Performance: Strategies and Tactics; Addison Wesley, 2000.

[Krall (1998)] Krall, A.: "Efficient javavm just-in-time compilation"; International Conference on Parallel Architectures and Compilation Techniques; 205–212; 1998.

[Menezes et al. (c1997)] Menezes, A. J. , Van Oorschot, P. C., Vanstone, S. A.: Handbook of applied cryptography; CRC Press series on discrete mathematics and its applications; CRC Press, Boca Raton, c1997; includes bibliographical references (p. 703-754) and index.

[SUN Microsystems (1997)] SUN Microsystems: "Java Native Interface Specification"; Available online at: http://java.sun.com/j2se/1.5.0/docs/guide/jni/spec/jniTOC.html (1997).

[Ortiz (2002)] Ortiz, E.: "Introduction to ota application provisioning"; Technical report; SUN Developer Network (2002); article available at: http://developers.sun.com/mobility/midp/articles/ota/.

[JSR 139 Process (2004)] SUN Community Process JSR 139: "*J2ME(TM) Connected Limited Device Configuration (CLDC) Specification 1.1 Final Release*"; Specification available at: http://jcp.org/aboutJava/communityprocess/final/jsr139/index.html (2004).

[Sun Microsystems (2002)] Sun Microsystems: "K Native Interface (KNI)"; Technical report; 4150 Network Circle Santa Clara, California 95054 (2002).

[TCG-Mobile-Phone-Working-Group (2007)] TCG-Mobile-Phone-Working-Group: "*TCG Mobile Trusted Module Sepecification Version 1 rev. 1.0*"; Specification available online at: https://www.trustedcomputinggroup.org/specs/mobilephone/tcg-mobile-trusted-modul e-1.0.pdf (2007).

[Yellin, Lindholm (1999)] Yellin Frank, Lindholm Tim: "*The Java Virtual Machine Specification Second Edition*"; Available online at: http://java.sun.com/docs/books/jvms/second_edition/html/VMSpecTOC.doc.html (1999).