Generating Standard DSA Signatures Without Long Inversion

Arjen K. Lenstra

Citibank, N.A., 4 Sylvan Way, Parsippany, NJ 07054, U.S.A.
E-mail: arjen.lenstra@citicorp.com

Abstract. We show how the generation of a random integer k modulo q and the subsequent computation of $k^{-1} \bmod q$ during the signature phase of the NIST digital signature algorithm (DSA) can be replaced by the simultaneous generation of a pair $(k, k^{-1} \bmod q)$. The k generated by our method behaves as an unpredictable integer modulo q that cannot, as far as we know, be efficiently distinguished from a truly randomly generated one. Our approach is useful for memory-bound implementations of DSA, because it avoids modular inversion of large integers. It is different from the inversion-free but non-standard method from [10], thus avoiding possible patent issues and incompatibility with standard DSA signature verification implementations. Another application of our method is in the 'blinding' operation that was proposed by Ron Rivest to foil Paul Kocher's timing attack on RSA, or in any other situation where one needs a random number and its modular inverse.

1 Introduction

Each user of the NIST digital signature algorithm (DSA) has a public key (p, q, g, y) and a corresponding secret key x. Here p is a $(512 + i \cdot 64)$-bit prime for some $i \in \{0, 1, \ldots, 8\}$, $g \in (\mathbf{Z}/p\mathbf{Z})^*$ is an element of order q for a 160-bit prime divisor q of $p-1$, and $y \in (\mathbf{Z}/p\mathbf{Z})^*$ and $x \in \{1, 2, \ldots, q-1\}$ are such that $y = g^x$ (cf. [11]). Powers of g are represented in the usual way by integers in $\{1, 2, \ldots, p-1\}$.

The signature of a message m consists of a pair (r, s) such that $r \equiv t \bmod q$ with $t = g^k \in (\mathbf{Z}/p\mathbf{Z})^*$ and $s \equiv (H(m) + xr)/k \bmod q$ for a randomly chosen $k \in \{1, 2, \ldots, q-1\}$ and a 160-bit hash $H(m) \in \mathbf{Z}$ of the message m. It is required that k be chosen such that $s \not\equiv 0 \bmod q$. Both r and s must be represented by integers in $\{1, 2, \ldots, q-1\}$. To verify a signature (r, s) for a message m, the verifier computes $u \equiv H(m)/s \bmod q$, $v \equiv r/s \bmod q$, and $w = g^u y^v \in (\mathbf{Z}/p\mathbf{Z})^*$, and accepts the signature if $r \equiv w \bmod q$.

The computationally most intensive part of the signature and verification phases of DSA are the 'modular exponentiations' modulo the fixed modulus p: one modular exponentiation during the signature generation to compute t, and

two during the verification [1] to compute w. Furthermore, there are a few modular multiplications modulo the fixed modulus q, and one inversion modulo q in the signature and verification phase each.

In this note we restrict ourselves to the computation that has to be carried out by the signer. We assume that the signer is computationally weak. Our purpose is to implement the signature phase of DSA in a time and memory efficient fashion. In particular we consider the computation of $k^{-1} \bmod q$ for a randomly selected k. Notice that the secret key x can be computed if k is exposed for any signature, or if the same k is used more than once by the same signer.

In a DSA implementation where modular arithmetic is implemented using standard arithmetic (i.e., first the product is computed and next its remainder modulo the modulus), it should be possible to implement the extended Euclidean algorithm with little extra overhead—either in software or on-chip, and possibly with inclusion of 'Lehmer's trick' (cf. [5: 4.5.2]—throughout this note we refer to this method as 'Lehmer's inversion'). This would make it possible to compute $k^{-1} \bmod q$ at a very small fraction of the cost of a modular exponentiation with modulus p, using only a few divisions on integers of at most 160 bits.

Many implementations of cryptographic protocols, however, do not use standard arithmetic but *Montgomery arithmetic*. If the modulus is fixed, this allows division-free modular multiplication and exponentiation [9]. Compared to regular modular arithmetic it is often faster and implementations require less code. Montgomery arithmetic does, however, not offer a convenient way to directly implement Lehmer's inversion. Nevertheless, since

$$k^{-1} \equiv k^{q-2} \bmod q, \tag{1.1}$$

Montgomery arithmetic can, in principle, also be used for the computation of $k^{-1} \bmod q$ in cases where, as in DSA, the modulus q is prime. Thus, if only Montgomery arithmetic is available, $k^{-1} \bmod q$ can be computed at about 1/40th to 1/10th (depending on the relative sizes of p and q) of the cost of a modular exponentiation with modulus p and with hardly any memory overhead. Although this is reasonably fast, it is considerably slower than Lehmer's inversion.

A faster solution was proposed in [10]: replace the signature (r, s) by the triple (r, a, b) with $a = (H(m) + xr) \cdot d \bmod q$ and $b = k \cdot d \bmod q$ for some randomly chosen $d \in \{1, 2, \ldots, q-1\}$, thus avoiding computation of any modular inverses by the signer. The verifier can then compute $1/s \bmod q$ as $b/a \bmod q$. So, both in the original DSA and in this variation the verifier has to carry out one inversion modulo q. Despite its simplicity and elegance this method has three disadvantages. In the first place, the protocol is slightly different from the standard DSA protocol, which might cause incompatibilities with verifiers that follow the standard. In the second place, this method requires twice the number of random bits required by standard DSA. Whether or not these two

[1] The two modular exponentiations during the verification can be performed simultaneously at the cost of about 5/4th of a single modular exponentiation (cf. [16]).

disadvantages create a problem in practice depends on the intended application of the implementation. Finally, usage of this method might require a license because it is, to the best of our knowledge, patented.

The purpose of this note is to provide an alternative method to generate k and $k^{-1} \bmod q$. Except for the way k and $k^{-1} \bmod q$ are generated, our method follows the standard DSA protocol, does not require division of 'random' integers of about 160 bits, and is suitable for memory-bound-chip implementations. With the proper parameter selection its speed is competitive with Lehmer's inversion. Our method is thus substantially faster than the method based on (1.1), but it uses slightly more memory. It uses modular multiplication with a fixed modulus q, for which Montgomery arithmetic can be used.

Although our method was developed specifically for application in DSA, it can be used in any application where a random number and its modular inverse are needed. As an alternative example we mention the 'blinding' operation for RSA encryption or signature generation. As shown by Paul Kocher (cf. [7]), timing information of the 'secret computation' $y = x^d \bmod n$ for several known x's (where n is the composite public modulus and d is the secret exponent corresponding to the public exponent e, i.e., $e \cdot d \equiv 1 \bmod \varphi(n)$) might reveal d.

To foil this so-called 'timing attack', Ron Rivest suggested first to blind x as $x' = xk^e \bmod n$ for a random integer k, next to replace the secret computation by $y' = x'^d \bmod n$, and finally to compute y as $y'k^{-1} \bmod n$ (cf. [4]). The pair $(k, k^{-1} \bmod n)$ necessary for the blinding can be obtained using straightforward application of Lehmer's inversion, using the proper variation of (1.1) (namely $k^{-1} \equiv k^{ed-1} \bmod n$), or by means of the method presented in this note.

We assume that integer arithmetic modulo some radix $R = 2^B$, for some reasonably large B, can be carried out efficiently. The value $R = 2^{32}$ would certainly suffice, and $B = 16$ might also be enough; $B = 8$ is probably too small. Our method requires division of integers of at most $160 + B$ bits by integers of at most B bits, multiplication of integers of at most 160 bits by integers of at most B bits, and the extended Euclidean algorithm on integers of at most B bits. The additional memory required for these three functions should be small compared to what is needed for Lehmer's inversion of 160-bit numbers.

In Section 2 we describe our method for the simultaneous computation of k and $k^{-1} \bmod q$, in Section 3 we discuss the unpredictability of the resulting k and $k^{-1} \bmod q$, and in Section 4 we compare the run times and code sizes of software implementations of our method and of the methods discussed above.

2 Simultaneous generation of k and $k^{-1} \bmod q$

Our method to simultaneously generate k and $k^{-1} \bmod q$ is based on the following three simple observations:

1. the inverse modulo q of a random number t of at most B bits can easily be computed;

2. a product modulo q of sufficiently many such t^z's, with each z randomly selected from $\{-1, 1\}$, and the inverse modulo q of this product can both easily be computed; and
3. neither such a product nor its inverse can efficiently be distinguished from a random number modulo q (cf. Section 3).

This leads to the following algorithm for the simultaneous computation of k and $k^{-1} \bmod q$.

2.1 **Algorithm.** Initialize k and $\bar{k}$ both as 1 (cf. Remark 2.2). Let $m > 2b$, for appropriately chosen positive integers b and m (cf. Section 3). For $i = 1, 2, \ldots, m$ in succession perform steps (1) through (4):

1. Select a random integer $t \in \{2, 3, \ldots, R-1\}$ and compute $q_t \equiv -q \bmod t$ with $q_t \in \{1, 2, \ldots, t-1\}$, using one 160-bit by B-bit division.
2. Attempt to compute $q_t^{-1} \bmod t$ using the extended Euclidean algorithm on integers of at most B bits (cf. Remark 2.4); return to Step (1) upon failure (i.e., if t and q_t are not co-prime), proceed to Step (3) otherwise.
3. Compute $t^{-1} \bmod q$ as $(q \cdot q_t^{-1} + 1)/t$, using one 160-bit by B-bit multiplication, and one $(160 + B)$-bit by B-bit division.
4. Let $s = 0$ if $i \leq b$, let $s = 1$ if $b < i \leq 2b$, and let s be randomly selected from $\{0, 1\}$ if $2b < i \leq m$. Replace k by $k \cdot t^{1-2s} \bmod q$ and $\bar{k}$ by $\bar{k} \cdot t^{2s-1} \bmod q$; this takes two modular multiplications modulo q.

Return k and $\bar{k} = (k^{-1} \bmod q)$.

Correctness. Because $q_t \equiv -q \bmod t$ we have that t divides $q \cdot q_t^{-1} + 1$, i.e., that $(q \cdot q_t^{-1} + 1)/t$ is an integer. This implies that $t \cdot ((q \cdot q_t^{-1} + 1)/t)$ equals 1 modulo q, so that $t^{-1} \bmod q$ is computed correctly. The correctness of the remainder of the algorithm follows immediately from that fact that $1 - 2s$ equals either 1 or -1.

2.2. **Remark.** In Montgomery arithmetic modulo q an integer x is represented by $\hat{x} = x \cdot R_q \bmod q$, for some appropriate power R_q of R. Consequently, the Montgomery product $\hat{z}$ of the Montgomery numbers $\hat{x} = x \cdot R_q \bmod q$ and $\hat{y} = y \cdot R_q \bmod q$ equals $z \cdot R_q \bmod q$ with $z \equiv x \cdot y \bmod q$. To find $\hat{z}$ given $\hat{x}$ and $\hat{y}$, it suffices to compute the ordinary product of $\hat{x}$ and $\hat{y}$, followed by a division by R_q modulo q. The latter division can be done using shifts, which is one of the reasons that Montgomery arithmetic may be preferable to regular modular arithmetic modulo q.

If the regular modular multiplication modulo q in the computation of k and $\bar{k}$ is replaced by Montgomery multiplication, the results will be $k \cdot R_q^{-m} \bmod q$ and $k^{-1} \cdot R_q^{-m} \bmod q$ instead of k and $k^{-1} \bmod q$. At least one of these results will have to be (Montgomery) multiplied by an appropriate power of R_q to get a correct result. This can easily be achieved by initializing k and $\bar{k}$ as appropriate powers of R_q. Which power depends on the rest on the implementation. For on-chip implementation the powers of $R_q \bmod q$ can be 'hard-wired' during the personalization of the chip, at the time the other fixed DSA-related values, like q and p, are initialized as well.

Because one of the multipliers (t) in Step (4) is at most B bits long, one of the two modular multiplications in Step (4) can be replaced by a 'partial' Montgomery multiplication: compute the ordinary product, and next divide by R (instead of R_q) modulo q. This is about $160/B$ times faster than a 'full' Montgomery multiplication.

2.3. **Remark.** In practice the bits s in Step (4) can be randomly selected for all i, as long as s is at least b times equal to 0 and at least b times equal to 1. See also Section 3.

2.4. **Remark.** If t is odd, Step (2) can be carried out using the fast division-free binary version of the extended Euclidean algorithm (cf. Appendix). Because for B-bit integers the binary method is considerably faster than the standard method, we only use odd t's in our implementation.

3 Security Considerations

Obviously, the k's as generated by Algorithm (2.1) are not ordinary random numbers in $\{1, 2, \ldots, q-1\}$ as is required in DSA. The danger of replacing truly random values by values that have more 'structure' or otherwise certain known arithmetic properties is well known. This is for instance illustrated by a series of four papers, alternatingly by Schnorr and de Rooij—with both Schnorr's original method to efficiently generate x^k's modulo q for random looking k's (cf. [13]) and his fix (cf. [14]) broken by de Rooij's subsequent attacks (cf. [1, 2]).

If b and m are relatively small, it is in principle possible to distinguish k's generated by our method from truly randomly generated ones. For instance, with $B = 32$, $b = 2$, and $m = 5$, one may try and divide k and its inverse by the product of two B-bit integers, and check if the inverse modulo q of any of the results is a small (i.e., $3B$-bit) number that is the product of three smaller ones. The efficiency of this 'attack' is questionable, however, and even if it were efficient it would be of limited use because the value for k is not revealed in DSA. The obvious next question is whether a signature (r, s) computed with a truly randomly generated k can be distinguished from a signature computed with a k generated by Algorithm (2.1), and whether this would undermine the security of the signature. We are not aware of a method to do this in a feasible amount of time, even for $B = 32$, $b = 2$, and $m = 5$. Nevertheless, we encourage (and look forward to) cryptanalysis of our method, strongly advise against its application before it has extensively been scrutinized, and encourage prospective users to consult future cryptology proceedings.

With $B = 32$ we found that, as long as the total number of random bits used to generate the t's is at least [2] 160 and $b \geq 2$, the k as generated by Algorithm (2.1), using the random number generator provided in LIP, the author's package

[2] Here we only count the bits for the t's for which t and q_t are co-prime. With properly selected t's the survival rate of the t's can be made very high, so that only a small number of random bits is wasted.

for long integer arithmetic (cf. [6]), cannot efficiently be distinguished from 160-bit values directly generated by that same generator, using several standard statistical tests: Marsaglia's diehard test (cf. [8]), and an efficient fixed size subset sum test developed by S. Rajagopalan and R. Venkatesan (cf. [12]) based on Goldreich-Levin hard core bits (cf. [3]) and U. Vazirani's subset sum test (cf. [15]).

The selection of the t's can be done in many ways. Some care should be taken that they do not have too many factors in common. To increase the probability that this happens they could for instance be chosen congruent to 1 modulo 30. This choice also increases the survival rate of the t's in Step (2) of Algorithm (2.1).

Even though the executions of Step (4) of Algorithm (2.1) require $m - 2b$ additional random bits, the approach sketched there is probably better than assigning half of the t's to k and the other half to $\bar{k}$ because it makes it harder to predict how many 'small factors' contributed to k and how many to $k^{-1} \bmod q$. Also the powers of R_q might be distributed in some random fashion over k and $k^{-1} \bmod q$ (cf. Remark 2.2).

4 Comparison with other Methods

Run times. We implemented Algorithm (2.1) using the method from the Appendix and the long integer package LIP (cf. [6]). Averaging over 100 different 160-bit primes q and 100 k's per q, we found that Algorithm (2.1) with $B = 32$, $b = 2$, and $m = 5$ took 0.8 milliseconds on a Sparc 10/51 workstation. Note that this choice allows about 160 random bits for the choice of the t's and thus satisfies the requirements mentioned in Section 3. For each of the resulting k's we also computed $k^{-1} \bmod q$ using two other methods: Lehmer's inversion [3] took 1 millisecond per k, and the exponentiation method based on (1.1) took 11.7 milliseconds per k. The 0.8 and 11.7 millisecond timings can be improved slightly using Montgomery arithmetic. Obviously, the run time of Algorithm (2.1) is linear in m; for the more 'secure' choice $b = 5$, $m = 15$, for example, we found that Algorithm (2.1) takes on average 2.3 milliseconds.

From these run times we conclude that for DSA Algorithm (2.1) can be made competitive with Lehmer's inversion, but that, if the latter is available, it hardly makes sense to use Algorithm (2.1). In an environment where this is not the case and where Montgomery arithmetic is used, however, Algorithm (2.1) is substantially faster than the 'standard' solution based on the use of (1.1). Because the run time of the latter method is proportional to $(\log q)^3$ and the other two are only quadratic in $\log q$ (because b and m are proportional to $\log q$), the advantage of Algorithm (2.1) compared to (1.1) becomes more apparent for larger q. For instance, for 512-bit primes q, Algorithm (2.1) with

[3] Lehmer's inversion applied to sufficiently large integers is more efficient than the method from the Appendix applied to large integers.

$b = 6$ and $m = 16$ takes on average 7 milliseconds, Lehmer's inversion needs 4 milliseconds, and the method based on (1.1) takes 210 milliseconds.

Code sizes. In a scenario as in DSA where regular modular multiplication and exponentiation or Montgomery multiplication and exponentiation are available, computing the inverse using the exponentiation method based on (1.1) requires no additional code. Algorithm (2.1) requires, as mentioned in Section 1, code for division of $(160+B)$-bit integers by B-bit integers (this is most likely already available if regular modular arithmetic is used for the remainder of DSA), multiplication of 160-bit integers by B-bit integers (already available either in regular or Montgomery arithmetic), and the extended Euclidean algorithm on integers of at most B bits. In LIP this would lead to an overhead of 100 lines of code for the division and 60 lines for the inversion (or only 60 if regular modular arithmetic is used). For Lehmer's inversion one needs full-blown long integer division, and the inversion code. In LIP this would lead to an overhead of 150 lines for the division and 200 lines for Lehmer's inversion (or only 200 lines if regular arithmetic is used). All these figures are *very* rough upper bounds as they include formatting, defines, and many statements to make the code generally applicable but that are not all needed in any specific application.

Which of the three alternatives is preferable depends on the actual implementation and other system characteristics, requirements, and specifications. We have not been able to carry out any comparisons using smartcards.

Acknowledgments. Acknowledgments are due to S. Haber, R. Venkatesan, and S. Rajagopalan for their assistance. The work presented in this paper was done at Bellcore.

References

1. P. de Rooij, *On the security of the Schnorr scheme using preprocessing*, Advances in Cryptology, Eurocrypt'91, Lecture Notes in Comput. Sci. **547** (1991) 71–80.
2. P. de Rooij, *On Schnorr's preprocessing for digital signature schemes*, Advances in Cryptology, Eurocrypt'93, Lecture Notes in Comput. Sci. **765** (1994) 435–439.
3. O. Goldreich, L. A. Levin, *Hard core bits for any one way function*, 22nd Annual ACM symposium on theory of computing (1990); J. Symbolic Logic **58** (1993) 1102–1103.
4. B. Kaliski, *Timing attacks on Cryptosystems*, RSA Laboratories' Bulletin, Number 2, January 1996.
5. D. E. Knuth, *The art of computer programming*, volume 2, *Seminumerical algorithms*, second edition, Addison-Wesley, Reading, Massachusetts, 1981.
6. A. K. Lenstra, *LIP, Long integer package*, available by anonymous `ftp` from `ftp.ox.ac.uk:/pub/math/freelip/freelip_1.0.tar.gz`.
7. J. Markoff, *Secure digital transactions just got a little less secure*, New York Times, December 11, 1995.
8. G. Marsaglia, *Diehard randomness tests package*, available by e-mail from the author (`geo@stat.fsu.edu`).
9. P. L. Montgomery, *Modular multiplication without trial division*, Math. Comp. **44** (1985) 519–521.

10. D. Naccache, D. M'Raïhi, D. Raphaeli, S. Vaudenay, *Can D.S.A. be improved — complexity trade-offs with the digital signature standard*, Preproceedings Eurocrypt'94 (1994) 85–101.
11. NIST, *A proposed federal information processing standard for digital signature standard (DSS)*, Federal Register **56** (1991) 42980–42982.
12. S. Rajagopalan, R. Venkatesan, work in progress.
13. C. P. Schnorr, *Efficient identification and signatures for smart cards*, Advances in Cryptology, Crypto'89, Lecture Notes in Comput. Sci. **435** (1990) 239–251.
14. C. P. Schnorr, *Efficient signature generation by smart cards*, Journal of Cryptology **4** (1991) 161–174.
15. U. Vazirani, Randomness, adversaries and computations, Ph.D. thesis, UC Berkeley, 1986.
16. S.-M. Yen, C. Laih, A. K. Lenstra, *A note on multi-exponentiation*, IEE Proceedings, Computers and digital techniques **141** (1994).

Appendix

Because this method does not seem to be generally known, we review an efficient and division-free method to compute $n^{-1} \bmod p$ for odd p and n with $0 < n < p$ and p and n co-prime:

Binary extended Euclidean algorithm. Let $n_1 = n$, $m_1 = 1$, $n_2 = p$, and $m_2 = 0$. Throughout the algorithm we have that $m_i \cdot n \equiv n_i \bmod p$, for $i = 1, 2$. First, as long as n_1 is even, replace n_1 by $n_1/2$ and m_1 by $m_1/2 \bmod p$ (because p is odd, dividing m_1 by 2 modulo p can be done either by right-shifting m_1 if m_1 is even, or by right-shifting $m_1 + p$ if m_1 is odd). Next, perform steps (1) through (6), until the algorithm terminates:

1. If n_1 equals 1, return $m_1 = n^{-1} \bmod p$ and terminate;
2. If n_1 equals n_2 then n and p are not co-prime: report failure and terminate;
3. If $n_1 < n_2$ swap n_1 and n_2 and swap m_1 and m_2;
4. Replace n_1 by $n_1 - n_2$ and replace m_1 by $m_1 - m_2$ (note that the resulting n_1 is even);
5. Replace n_1 by $n_1/2$ and m_1 by $m_1/2 \bmod p$;
6. If n_1 is even return to Step (5), otherwise return to Step (1).

This binary extended Euclidean algorithm is much more efficient than the standard extended Euclidean algorithm when applied to B-bit integers. As mentioned in the footnote in Section 4, however, it is less efficient than Lehmer's inversion when applied to 160-bit or larger integers. Note that for B-bit integers Lehmer's inversion is identical to the standard extended Euclidean algorithm.