GOST 34.10 – A brief overview of Russia's DSA

Markus Michels[1] · David Naccache[2] · Holger Petersen[3]

Abstract

GOST 34.10 is Russia's DSA. Just as it's U.S. counterpart, GOST is an ElGamal like signature scheme used in Schnorr mode. It is close to NIST DSA in many aspects. In this paper we will first overview GOST 34.10 and discuss the three main differences between the two algorithms.

- GOST's principal design criterion doesn't seem to be computational efficiency: The algorithm is 1.6 times slower than the DSA and produces 512 bit signatures. This is mainly due to the usage of the modulus q which is at least 254 bit long. During verification, modular inverses are computed by exponentiation (while the Extended Euclidian algorithm is roughly 100 times faster for this parameter size) and the generation of the public parameters is much more complicate than in the DSA. This choice of the parameters makes GOST 34.10 very secure.
- GOST signers don't have to generate modular inverses as the basic signature equation is $s = xr + mk \pmod q$ instead of $s := (m + xr)/k \pmod q$.
- GOST's hash function (the Russian equivalent of the SHA) is the standard GOST 34.11 which uses the block cipher GOST 28147 (partially classified) as a building block. The hash function will be briefly described.

1. Introduction

Since its invention by Diffie and Hellman in 1976 [4], many realizations for digital signatures have been proposed. The most famous among them are certainly the RSA [20], ElGamal [5] Schnorr's [21] and the Fiat-Shamir scheme [7].
Several organizations proposed different digital signature standards. For example, the ISO/IEC 9796 specifies the RSA for digital signatures [15, 11], the American National Standard X9.30-199X RSA and the ElGamal signature scheme [1], the French banking community has also standardized on RSA [6]. The National Institute of Standardization and Technology (NIST) proposed a modification of the ElGamal and Schnorr scheme as

[1]Theoretical Computer Science and Information Security, University of Technology Chemnitz-Zwickau, 09111 Chemnitz, Germany. E-mail: mmi@informatik.tu-chemnitz.de
This work was done while visiting Gemplus PSI, Sarcelles, France.
[2]Gemplus Card International, Crypto Team, 95200 Sarcelles, France.
E-mail: 100142.3240@compuserve.com
[3]Laboratoire d'informatique, Ecole Normale Supérieure, 45, rue d'Ulm, 75005 Paris, France.
E-mail: petersen@dmi.ens.fr
This work was done while visiting Gemplus PSI, Sarcelles, France.

their Digital Signature Standard (DSS) [17] and signature schemes based on elliptic curves are currently being considered as an IEEE standard [16].
In 1994 the Russian Federation issued the digital signature standard GOST 34.10 – 94 [9]. This standard covers the system initialization, key generation, signature generation and verification. It also uses the hash function standardized under the reference GOST 34.11 – 94 [10]. This hash function is based on the, partially classified, block cipher GOST 28147 – 89 [8]. The signature generation and verification procedures used by the standard are a modification of the DSA. This modification has also been proposed in the literature [12, 18].
We use the following notation which is adopted from the standard. B^* is the set of all finite words in the alphabet $B = \{0, 1\}$ and h the hash function representing the message m in the word $h(m) \in \{0, 1\}^{256}$.

2. Initialization

During the initialization of the system the trusted authority generates two primes p, q with $q|(p-1)$ and the generator α of order q. These values are fixed for the whole session. For the generation of these parameters a constant c is used. Here we review one of the four algorithms for generating the large primes described in the standard (the other three are very similar to this one).

2.1 Prime generation

This procedure determines a prime p with length $t \geq 17$ bits and a prime divisor q, $q|(p-1)$ of length $\lfloor t/2 \rfloor$ bits. It is realized using the linear congruence $x_n := 19381 \cdot x_{n-1} + c$. It is necessary to fix a number $x_0 \in [1 : 2^{16} - 1]$ and an odd number $c \in [1, 2^{16} - 1]$. The calculation procedure contains the following.

1. $y_0 := x_0$,
2. Calculate the sequence of numbers $t_0, t_1, \ldots, t_s$ according to the rule: $t_0 := t$, if $t_i \geq 17$ then $t_{i+1} := \lfloor t_i/2 \rfloor$ otherwise $s := i$,
3. Find the smallest prime p_s of length t_s bits, that is
$$\forall p, p \text{ is a prime}, |p| = t_s : p_s < p,$$
4. $m := s - 1$,
5. Calculate $r_m := \lceil t_{m+1}/16 \rceil$,
6. Calculate the sequence $y_1, \ldots, y_{r_m}$ according to the recursive rule:
$$y_{i+1} := (19381 \cdot y_i + c) \pmod{2^{16}}.$$
7. Calculate $Y_m := \sum_{i=0}^{r_m - 1} y_i \cdot 2^{16 \cdot i}$,
8. $y_0 := y_{r_m}$,
9. $N = \lceil 2^{t_m - 1}/p_{m+1} \rceil + \lfloor 2^{t_m - 1} \cdot Y_m/(p_{m+1} \cdot 2^{16 \cdot r_m}) \rfloor$. If N is odd then $N := N + 1$,
10. $k := 0$,
11. Calculate $p_m = p_{m+1} \cdot (N + k) + 1$,
12. If $p_m > 2^{t_m}$ go to step 6,
13. Verify the conditions:
$$2^{p_{m+1}(N+k)} \pmod{p_m} = 1$$
$$2^{(N+k)} \pmod{p_m} \neq 1,$$
If one of this conditions is not realized then $k := k + 2$ and pass to step 11. If both conditions are realized then $m := m - 1$,
14. If $m \geq 0$, pass to step 5, If $m < 0$, then $p := p_0$ and $q := p_1$.

In the standard there is a similar procedure to create a number p of size bigger than 33 bits, which uses the recursive rule $y_{i+1} := (97781173 \cdot y_i + c) \pmod{2^{32}}$ instead.

2.2 Generator

This procedure permits to determinate the generator α after p and q have been determined.

1. Choose a random number $d \in \mathbf{Z}_p^*$.
2. Calculate $f := d^{(p-1)/q} \pmod{p}$.
3. If $f = 1$, then go to step 1, If $f \neq 1$ then $\alpha := f$.

3. Key generation

A trusted authority chooses a large prime p, with $2^{509} < p < 2^{512}$ or $2^{1020} < p < 2^{1024}$, and a prime divisor q, $2^{254} < p < 2^{256}$, with $q|(p-1)$. Every user chooses a secret key $x \in \mathbf{Z}_q$ and computes his related public key by $y := \alpha^x \pmod p$. To guarantee the security of the system, the public keys of the user must be certified by a trusted authority although this is not explicitly mentioned in the standard.

4. Signature generation

The signature generation for the message $m \in B^*$ is done by the following algorithm.

1. Calculate $h(m)$, the hash value of the message m, using the GOST 34.11 hash function. If $h(m) \equiv 0 \pmod q$ then set $h(m) := 0^{255}1$.
2. Create the random number $k \in \mathbf{Z}_q^*$.
3. Calculate the two values $r' := \alpha^k \pmod p$ and $r := r' \pmod q$. If $r = 0$, pass to step 2 and create another random number k.
4. Using the signer's secret key x, calculate the value:

$$s := (x \cdot r + k \cdot h(m)) \pmod q.$$

   If $s = 0$, pass to step 2.

The signature on the message m is the tuple (r, s), sent to the verifier.

5. Signature verification

The verifier checks the authenticity of the message by checking the validity of the signature. This is possible, if he knows the public key of the signer. The signature verification contains the following steps.

1. Verify the conditions $0 < s < q$ and $0 < r < q$. If one of these conditions is not satisfied, the signature is not valid.
2. Calculate $h(m_1)$, the hash value of the received message m_1. If $h(m_1) \equiv 0 \pmod q$, then set $h(m_1) := 0^{255}1$.
3. The verification equation is

$$r \equiv (\alpha^{s \cdot h(m_1)^{-1}} \cdot y^{-r \cdot h(m_1)^{-1}} \pmod p) \pmod q. \tag{1}$$

It can be checked by the following steps:

a) Calculate the value $v := h(m_1)^{q-2} \pmod q$, which is the multiplicative inverse of $h(m_1) \pmod q$.

b) Compute the values $z_1 := s \cdot v \pmod q$ and $z_2 := (q - r) \cdot v \pmod q$.

c) Calculate the value $u := (\alpha^{z_1} \cdot y^{z_2} \pmod p) \pmod q$,

d) Verify the condition $r = u$. If this check succeeds, the message was signed by the signer.

6. Description of the hash function

6.1 Introduction

An important part of the GOST 34.10 – 94 is the hash function standardized in GOST 34.11 – 94 [10]. The use of a hash function is necessary because messages of arbitrary length should be efficiently signed. Furthermore, existential forging of a signature can be avoided. The GOST 34.11 – 94, the State Standard of the Russian Federation for hashing, is valid since January 1st, 1995. It uses the block cipher standardized in GOST 28147 – 89 [8], which is partially classified. This block cipher is used in the Electronic Code Book Mode (ECB) during the computation of the hash function. For a description of this block cipher (in english) we refer to the literature [3, 19].
We use the following notation to describe the standard. $\hat{A}$ is the positive integer corresponding to the binary record A ($A \in B^*$), $< N >_k := N \pmod{2^k}$, $V_k(2)$ means the set of all binary words of length k, $\oplus$ is the bitwise xor of equally long words, $\oplus'$: the addition $A \oplus' B := < \hat{A} + \hat{B} >_k$, $(|A| = |B| = k)$, $E_K(A)$ is the coding of the word $A \in V_{64}(2)$ with key $K \in V_{256}(2)$ using the GOST 28147 algorithm, H is the 256 bit start vector for hashing. $C||D$ is the concatenation of the words $C, D \in B^*$ and A^k the concatenation of k copies of the word $A \in B^*$.

6.2 Overview

The hash function $h : B^* \times V_{256}(2) \longrightarrow V_{256}(2)$ maps a message of arbitrary length and the 256 bit start vector H onto a 256 bit hash value. The hash function is iterative. Therefore we need a step hash function x, which is a mapping $x : V_{256}(2) \times V_{256}(2) \longrightarrow V_{256}(2)$ and the description of the iterative procedure for calculating the hash value h. For the step hash function x we need

- the generation of the 256 bit keys K_1, K_2, K_3, K_4,
- the encryption, which encrypts 64 bit subwords of the start vector H and the keys K_i $(i = 1, 2, 3, 4)$ using the GOST 28147 algorithm.
- the permutation of the encryption output.

Now we describe these three modules.

6.3 Key Generation for the step hash function

6.3.1 Subfunctions

We need the subfunctions $A : V_{256}(2) \longrightarrow V_{256}(2)$ and $P : V_{256}(2) \longrightarrow V_{256}(2)$ for key generation.

- $A(x_4||x_3||x_2||x_1) := (x_1 \oplus x_2)||x_4||x_3||x_2$, where x_i $(i = 1, 2, 3, 4)$ is a 64 bit number.
- $P(\xi_{32}, \ldots, \xi_1) := \xi_{\varphi(32)}|| \ldots ||\xi_{\varphi(1)}$ where ξ_i is a 8 bit number and

$$\varphi(i + 1 + 4(k - 1)) = 8i + k, i = 0, \ldots, 3, k = 1, \ldots, 8.$$

6.3.2 Key generation algorithm

For key generation we need the following input data:

- the 256 bit start vector H and

- the 256 bit message part M.

We further need the constants C_i ($i = 2, 3, 4$) with values

$$C_2 = C_4 = 0^{256} \text{ and } C_3 = 1^8 0^8 1^{16} 0^{24} 1^{16} 0^8 (0^8 1^8)^2 1^8 0^8 (0^8 1^8)^4 (1^8 0^8)^4.$$

For the key calculation the following algorithm is used:

1. $i := 1, U := H, V := M$,
2. $W := U \oplus V, K_1 = P(W)$,
3. $i := i + 1$,
4. If $i = 5$ then go to step 7 else go to step 5,
5. $U := A(U) \oplus C_i, V := A(A(V)), W := U \oplus V, K_i = P(W)$.
6. Go to step 3,
7. End.

The keys K_1, K_2, K_3 and K_4 are the output of the algorithm.

6.4 Encryption

In this step we use the 64 bit subwords h_i ($i = 1, 2, 3, 4$) of the start vector $H = h_4||h_3||h_2||h_1$ and the 256 bit keys K_i ($i = 1, 2, 3, 4$) as input and get $S = s_4||s_3||s_2||s_1$ with 64 bit numbers s_i ($i = 1, 2, 3, 4$) as output by computing

$$s_i = E_{K_i}(h_i), \text{ for } i = 1, 2, 3, 4.$$

6.5 Permutation

The permutation is the mapping $\psi : V_{256}(2) \longrightarrow V_{256}(2)$ defined by

$$\psi(\eta_{16}||\dots||\eta_1) = \eta_1 \oplus \eta_2 \oplus \eta_3 \oplus \eta_4 \oplus \eta_{13} \oplus \eta_{16}||\eta_{16}||\dots||\eta_2$$

where η_i ($i = 1, \dots, 16$) is a 16 bit number.

6.6 Step hash function

The 256 bit start vector H and a 256 bit message block M are the inputs of this procedure. Furthermore S is computed with inputs M and H using the key generation algorithm and the encryption described above. Therefore, for each 256 bit block of a larger message we get a new parameter S. The step hashing function x is defined as

$$x(M, H) := \psi^{61}(H \oplus \psi(M \oplus \psi^{12}(S))),$$

with S determined by M and H.

6.7 Iterative hash function

Now we are ready to describe the iterative hash function h. On input of the 256 bit start vector H and the message M of arbitrary length, the following algorithm is used.

1. $\Sigma := 0^{256}, L := 0^{256}$,
2. - If $|M| > 256$ then go to step 3,
   - $L :=< \hat{L} + |M| >_{256}$, $M' := 0^{256-|M|}||M$, $\Sigma := \Sigma \oplus' M'$,
   - $H := x(M', H)$, $H := x(L, H)$, $H := x(\Sigma, H)$,
   - End.
3. - Calculate the subword $M_s \in V_{256}(2)$ of the word $M = M_p||M_s$.
   - $H := x(M_s, H)$, $L :=< L + 256 >_{256}$, $\Sigma := \Sigma \oplus' M_s$, $M := M_p$,
   - Go to step 2.

The 256 bit value H in step 2 is the output of this algorithm and of the hash function $h(M)$.

7. Discussion of GOST 34-10

7.1 Security consideration

The security consideration concerns only the signature algorithm and its verification equation. The security analysis can be divided into three possible attacks [5]. In the first attack, an attacker wants to get the secret key of the signer. The second attack tries to forge a signature on almost all messages m using all known public parameters. In the third attack just a signature of a random message m should be forged.

1. *Total break* (Computing the secret key x):
   As the signature equation contains two unknown parameters x and k, a forger can't get any information about x without knowing k. Thus the calculation of x seems to be as difficult as the computation of the discrete logarithm $\log_\alpha(y_A)$.

   To avoid chosen plaintext attacks, the signer shouldn't sign the message $m \equiv 0 \pmod q$. This was detected by the standard designers who set $m := 1$ in this case. Otherwise, every verifier can compute the secret key x: The signature equation is $s \equiv xr + km \pmod q$. If $m \equiv 0$ then this simplifies to $s \equiv xr \pmod q$. From this x can be computed if $\gcd(r, p-1) = 1$.

2. *Universal forgery of messages:*
   To forge a message universally, the forger can first randomly choose an integer r, then attempts to find s or vice versa. In the first method, the forger must solve the discrete logarithm of
   $$y_A^r \cdot r^m \pmod p.$$
   In the second one, the attacker has to solve the equation
   $$d \equiv e^r \cdot r \pmod p, d, e \in \mathbf{Z}_p.$$
   This computation is extremly difficult, possibly even more difficult than solving the discrete logarithm problem. Of course if we can compute the discrete logarithm,

we can apply other direct attacks, so this question doesn't influence the security of the signature scheme itself. It might be possible, that the signature equation can be solved by simultaneous choice of r and s. Until today, it's an open problem, how this can be used for universal forgery in general. However, it must be guaranteed that the parameters are chosen with care. Note that the attack by Bleichenbacher [2] on the ElGamal scheme could be adopted, if q were chosen as a suitable composite. As q is chosen as a prime, the attack doesn't work. It must further be guaranteed that the generator is certified as otherwise the security of the scheme is lost [22]. Additionally, it might be possible to construct a useful collision by finding a suitable prime q, as shown in [22]. To avoid this attack it must be guaranteed by a certificate that q is really contructed by one of the prime generation algorithms.

3. *Existential forgery of messages:*
   The existential forgery of messages would be possible, if the message m is not hashed before signing. This is prevented in GOST 34.10 by signing the hash value $h(m)$ instead of m. Otherwise, the attacker chooses random values $u, w \in \mathbf{Z}_q^*$, computes the values $r := \alpha^u \cdot y_A^w \pmod p$, $m := -r \cdot w^{-1} \pmod q$ and $s := u \cdot m \pmod q$. Then (r, s) is a valid signature on the (random and probably useless) message m.

7.2 Performance considerations

The size of a secret key is $|q|$ bit and of the public key is $|p|$ bit. Furthermore the system parameter p with $|p|$ bit and the generator α with $|p|$ bit must be stored.
The signer needs only one exponentiation with a $|q|$ bit exponent, one reduction modulo q and two multiplication modulo q but no inverse for signature generation. The verifier has to compute three exponentiations, one modulo q with a $|q|$ bit exponent for computing an modular inverse and two modulo p with a $|q|$ bit exponent, and one multiplication modulo p. For example, with $|p| = 512$ and $|q| = 256$ by using Yen and Laih's method for computing exponentiations [23], we get the following result:
The computational effort of the signer is dominated by in average 308 multiplications modulo p for the one exponentiation. For verification the computational effort is dominated by 373 multiplications for the exponentiations modulo p and 308 multiplications for the exponentiation modulo q, what is comparable to 102 multiplications modulo p. Therefore, the amount of computation for verification is comparable to about 476 multiplications modulo p. The verifier can decrease the computational effort by about 100 multiplications if the modular inverse is computed by the Extended Euclidian Algorithm instead of exponentiation, what is about 100 times more efficient. However, this optimization is not suggested in the standard.
In comparison, in the DSA, with $|p| = 512$ and $|q| = 160$, the effort of the signer is dominated by the 200 multiplications modulo p for the exponentiation, if he uses the Extended Euclidian Algorithm for computing the modular inverse, which is then negligible. The effort of the verifier is dominated by computing the two exponentiations, what can be done with about 240 multiplications.
As a result, the DSA signature generation is about 1.6 times faster than GOST 34-10 (verification is about 2 times faster). This increased computational effort is caused by the bigger parameter q and the method suggested for computing modular inverses in the GOST 34-10.

8. Conclusion

We have briefly described the Russian signature and hash standard standard, GOST 34.10 – 94 and GOST 34.11 – 94 respectively. GOST's search space for brute force attacks is 2^{128} instead of 2^{80} in the DSA, which increases security. The use of exponentiation for computing modular inverses indicates that efficiency was not the standard designer's main motivation.
An advantage of GOST 34.10 – 94 over DSA is its inherit suitability to extended signature concepts, e.g. for blind signatures or multisignatures as already shown in [13, 14]. This property does not hold for the DSA.

References

[1] "Digital signatures for financial data using public key cryptography", ANSI Working draft of the American National Standard X9.30-199X, April, (1991).

[2] D.Bleichenbacher, "Generating ElGamal signatures without knowing the secret key", Lecture Notes in Computer Science 1070, Advances in Cryptology: Proc. Eurocrypt'96, Springer Verlag, (1996), pp. 10–18.

[3] C.Charnes, L.O'Connor, J.Pieprzyk, R.Saifavi-Naini, Y.Zheng, "Further comments on the Soviet Encryption Algorithm", University of Wollongong, Australia, Technical Report TR-9-94, (1994), 10 pages, available via anonymous FTP in directory ftp.cs.uow.edu.au/pub/papers/1994.

[4] W.Diffie, M.Hellman, "New directions in cryptography", IEEE Transactions on Information Theory, Vol. IT-22, No. 6, November, (1976), pp. 644–654.

[5] T.ElGamal, "A public key cryptosystem and a signature scheme based on discrete logarithms", IEEE Transactions on Information Theory, Vol. IT-30, No. 4, July, (1985), pp. 469–472.

[6] Échanges Télématiques Entre Les Banques et Leurs Clients, Standard ETBAC 5, Comité Français d'Organisation et de Normalisation Bancaires, (1989).

[7] A.Fiat, A.Shamir, "How to prove yourself: Practical solutions to identification and signature problems", Lecture Notes in Computer Science 263, Advances in Cryptology: Proc. Crypto '86, Springer Verlag, (1987), pp. 186–194.

[8] GOST 28147 – 89, "Systems of the information treatment. Cryptographic security. Algorithms of the cryptographic transformation", (1989).

[9] GOST 34.10 – 94, "Information Technology Cryptographic Data Security Produce and Check procedures of Electronic Digital Signature based on Asymmetric Cryptographic Algorithm", (1994).

[10] GOST 34.11 – 94, "Information Technology Cryptographic Data Security Hashing Function", (1994).

[11] L.C.Guillou, J.Quisquater, M.Walker, P.Landrock, C.Shaer, "Precautions taken against various potential attacks", Lecture Notes in Computer Science 473, Advances in Cryptology: Proc. Eurocrypt'90, Springer Verlag, pp. 465–473.

[12] P.Horster, H.Petersen, "Verallgemeinerte ElGamal Signaturen", Sicherheit in Informationssystemen, Proceedings der Fachtagung SIS'94, March, 10–11, 1994, Verlag der Fachvereine Zürich, (1994), pp. 89–106.

[13] P.Horster, M.Michels, H.Petersen, "Meta Message recovery and Meta Blind signature schemes based on the discrete logarithm problem and their applications", Lecture Notes in Computer Science 917, Advances in Cryptology: Proc. Asiacrypt '94, Springer Verlag, (1995), pp. 224–237.

[14] P.Horster, M.Michels, H.Petersen, "Meta-Multisignature schemes based on the discrete logarithm problem", Proc. IFIP/SEC'95, Cape Town, South Africa, May, (1995), pp. 128–142.

[15] ISO/IEC 9796, "Information Technology Security Techniques Digital signature scheme giving message recovery", July, (1991).

[16] A.Menezes, M.Qu, S.Vanstone, "IEEE P1363 Part 6: Elliptic Curve Systems", Draft, April 24, (1995).

[17] National Institute of Standards and Technology, Federal Information Process. Standard, FIPS Pub 186: Digital Signature Standard (DSS), (1994).

[18] K.Nyberg, R.A.Rueppel, "Message Recovery for Signature Schemes Based on the Discrete Logarithm", Pre-proceedings of Eurocrypt'94, Perugia, Italy, (1994), pp. 175–190, to appear in Lecture Notes in Computer Sciences, Springer Verlag.

[19] J.Pieprzyk, L.Tombak, "Soviet Encryption Algorithm", University of Wollongong, Australia, Technical Report TR-10-94, (1994), 9 pages, available via anonymous FTP in directory ftp.cs.uow.edu.au/pub/papers/1994.

[20] R.L.Rivest, A.Shamir, L.Adleman, "A method for obtaining digital signatures and public-key cryptosystems", Communications of the ACM, Vol. 21, No. 2, (1978), pp. 120–126.

[21] C.P.Schnorr, "Efficient identification and signatures for smart cards", Lecture Notes in Computer Science 435, Advances in Cryptology: Proc. Crypto '89, Springer Verlag, (1990), pp. 239–251.

[22] S.Vaudenay, "Hidden collisions on DSS", Lecture Notes in Computer Science 1109, Advances in Cryptology: Proc. Crypto'96, Springer Verlag, (1996), pp. 83–88.

[23] S.-M.Yen, C.-S.Laih, "The first cascade exponentiation algorithm and its applications to cryptography", Lecture Notes in Computer Science 718, Advances in Cryptology: Proc. Auscrypt '92, Springer Verlag, (1993), pp. 447–456.