Analysis of Software Realized DSA Algorithm for Digital Signature

Bojan R. Pajčin and Predrag N. Ivaniš

***Abstract*—In this paper the realization of one algorithm for digital signature, DSA (Digital Signature Algorithm), is presented. In the algorithm, for calculating the variables needed to generate public and private key, one-way hash function, SHA (Secure Hash Algorithm), is used. A method of realization of SHA and DSA is presented, and the time required to digitally sign messages of different sizes and time required to generate the keys are measured. The results are compared with the analogous result based on another software implemented system for digitally signing with hash function and RSA algorithm.**

***Index Terms*—Digital signature, hash function, public key encryption algorithms, software implementation.**

I. Introduction

PUBLIC-KEY algorithms (also called asymmetric algorithms) are designed so that the key used for encryption is different from the key used for decryption. Furthermore, the decryption key cannot (at least in any reasonable amount of time) be calculated from the encryption key [5, 6]. The algorithms are called "public-key" because the encryption key can be made public: A complete stranger can use the encryption key to encrypt a message, but only a specific person with the corresponding decryption key can decrypt the message. In these systems, the encryption key is often called the public key, and the decryption key is often called the private key. Sometimes, messages will be encrypted with the private key and decrypted with the public key; that protects the integrity of the sender (authenticity of the message). This is used in digital signatures.

A one-way hash function, often called compression function, message digest, fingerprint, cryptographic checksum, is central to modern cryptography. One-way hash functions are another building block for many protocols. Hash functions have been used in computer science for a long time. A hash function is a function, mathematical or otherwise, that takes a variable-length input string (called a pre-image) and converts it to a fixed-length (generally smaller) output string (called a hash value) [5]. A simple hash function would be a function that takes pre-image and returns a byte consisting of the XOR of all the input bytes. A one-way hash function is a hash function that works in one direction: It is easy to compute a hash value from pre-image, but it is hard to generate a pre-image that hashes to a particular value [5]. The hash function previously mentioned is not one-way: Given a particular byte value, it is trivial to generate a string of bytes whose XOR is that value. A good one-way hash function is also collision-free: It is hard to generate two pre-images with the same hash value. The hash function is public; there's no secrecy to the process. The security of a one-way hash function is its one-wayness. The output is not dependent on the input in any discernible way. A single bit change in the pre-image changes, on the average, half of the bits in the hash value. Given a hash value, it is computationally unfeasible to find a pre-image that hashes to that value.

The purpose of digital signature is to confirm the authenticity of the message content (proof that the message has not changed on the way between sender and recipient) and to ensure a guarantee of the sender identity. Base of digital signature is the contents of the message. Author (call him person A) using a certain cryptographic algorithms, firstly creates a fixed length record of its arbitrary length message, which fully reflects the content of the message (a hash value). After that, he performs certain operations over this record, using several other parameters and his secret key, and thus generates a digital signature that is sent along with the message. When the recipient (call him person B) receives a message with a digital signature, using the sender's public key (public key of person A) and the summary record of messages, which makes himself, and applying of certain operations as a result gets a number. Person B compares that number with the number who has received in the form of digital signature and thereby determines the authenticity of the message. Considering that operations in the message summary (hash value) use a secret key, nobody else can generate the digital signature except the person who sending the message (the only person A knows its secret key). This way, the recipient (person B) knowing the sender's public key (A) is a sure that just got a message from him, because the calculated value match only if

Ministry of Science and Technological Development of the Republic of Serbia has partially funded project TR32007 "Multiservice optical transport platform OTN10/40/100 Gbps with DWDM/ROADM and Carrier Ethernet functionalities." Part of results in this paper was presented at the 55th ETRAN conference, Banja Vrućica, 6-9. June 2011.

B. R. Pajčin is with the IRITEL A.D., Belgrade, Serbia (phone: 00 381 11 3073 451; fax: 00 381 3073 434; e-mail: bojan@iritel.com).

P. N. Ivaniš, is with Faculty of Electrical Engineering, University of Belgrade, Belgrade, Serbia (e-mail: predrag.ivanis@etf.rs).

the sender's public key corresponding to his secret key, which person A is used to generate a digital signatures.

In the next section will be presented the most frequently used hash functions today, Secure Hash Algorithm (SHA) [7], and briefly will be described its operation. It will be described operation of DSA algorithm [2], public key algorithm, which is part of the DSS standard. The above-mentioned hash function and DSA are software implemented and integrated in the system to digitally sign, and that system is the subject of this paper.

II. SHA

NIST (*National Institute of Standards and Technology*), along with the NSA (*National Security Agency*), designed the *Secure Hash Algorithm* (SHA) for use with the *Digital Signature Standard* (DSS). SHA produces a 160-bit hash.

First, the message is padded so that its length is just 64 bits short of being a multiple of 512. This padding is a single 1-bit added to the end of the message, followed by as many zeros as are required. Then, a 64-bit representation of the message's length (before padding bits were added) is appended to the result. These two steps serve to make the message length an exact multiple of 512 bits in length (required for the rest of the algorithm), while ensuring that different messages will not look the same after padding. After that, Five 32-bit variables are initialized as follows:

A = 0x67452301,
B = 0xefcdab89,
C = 0x98badcfe
D = 0x10325476,
E = 0xc3d2e1f0

The main loop of the algorithm then begins. It processes the message 512 bits at a time and continues for as many 512-bit blocks as are in the message. First the five variables are copied into different variables: *a* gets *A, b* gets *B, c* gets *C, d* gets *D,* and *e* gets *E*. The main loop has four rounds of 20 operations each. Each operation performs a nonlinear function on three of *a, b, c, d,* and *e,* and then does shifting and adding. SHA's set of nonlinear functions is:

$f_t(X,Y,Z) = (X \wedge Y) \vee ((\neg X) \wedge Z)$, for t = od 0 do 19
$f_t(X,Y,Z) = X \oplus Y \oplus Z$, for t = od 20 do 39
$f_t(X,Y,Z) = (X \wedge Y) \vee (X \wedge Z) \vee (Z \wedge Y)$, for t = od 40 do 59
$f_t(X,Y,Z) = X \oplus Y \oplus Z$, for t = od 60 do 79

Four constants are used in the algorithm [7]:

K_t = 0x5a827999, for t = od 0 do 19
K_t = 0x6ed9eba1, for t = od 20 do 39
K_t = 0x8f1bbcdc, for t = od 40 do 59
K_t = 0xca62c1d6, for t = od 60 do 79

(If you wonder where those numbers came from: 0x5a827999 = 21/2 /4, 0x6ed9eba1 = 31/2 /4, 0x8f1bbcdc = 51/2 /4, and 0xca62c1d6 = 101/2 /4; all times 232.)

The message block is transformed from 16 32-bit words (M_0 to M_{15}) to 80 32-bit words (W_0 to W_{79}) using the following algorithm:

$W_t = M_t$, for t = 0,1,…,15
$W_t = (W_{t-3} + W_{t-8} + W_{t-14} + W_{t-16}) <<< 1$, for t = 16,17,…,79

If t is the operation number (from 0 to 79), *Wt* represents the t-th sub-block of the expanded message, and $<<< s$ represents a left circular shift of s bits, then the main loop looks like [7]:

$TEMP = (a <<< 5) + f_t(b,c,d) + e + W_t + K_t$
$e = d$,
$d = c$,
$c = b <<< 30$,
$b = a$,
$a = TEMP$

Fig. 1 shows one operation. After all of this, *a, b, c, d,* and *e*

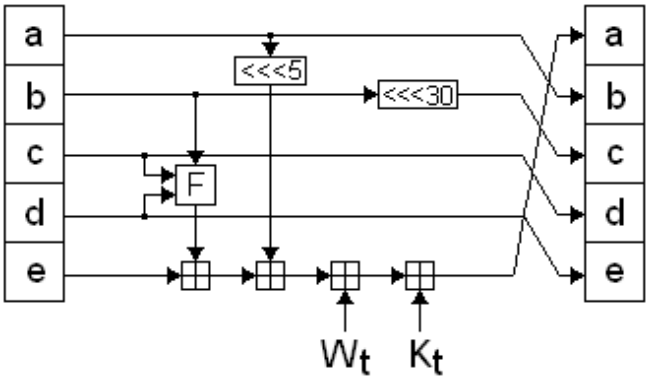

Fig. 1. Figure 1: One SHA operation.

are added to *A, B, C, D,* and *E* respectively, and the algorithm continues with the next block of data. The final output is the concatenation of *A, B, C, D,* and *E*

III. SECURITY OF SHA

There are two brute-force attacks against a one-way hash function. The first is the most obvious: Given the hash of message, $H(M)$, an adversary would like to be able to create another document, M', such that $H(M) = H(M')$. The second attack is more subtle: An adversary would like to find two random messages, *M,* and *M´,* such that $H(M) = H(M')$. This is called a collision, and it is a far easier attack than the first one.

The birthday paradox is a standard statistics problem. How many people must be in a room for the chance to be greater than even that one of them shares your birthday? The answer is 253. Now, how many people must there be for the chance to be greater than even that at least two of them will share the same birthday? The answer is surprisingly low: 23. With only 23 people in the room, there are still 253 different *pairs* of people in the room.

Finding someone with a specific birthday is analogous to the

first attack; finding two people with the same random birthday is analogous to the second attack. The second attack is commonly known as a birthday attack.

Assume that a one-way hash function is secure and the best way to attack it is by using brute force. It produces an m-bit output. Finding a message that hashes to a given hash value would require hashing 2^m random messages. Finding two messages that hash to the same value would only require hashing $2^{m/2}$ random messages.

Hash functions of 64 bits are just too small to survive a birthday attack. Most practical one-way hash functions produce 128-bit hashes. This forces anyone attempting the birthday attack to hash 2^{64} random documents to find two that hash to the same value, not enough for lasting security. NIST, in its Secure Hash Standard (SHS), uses a 160-bit hash value. This makes the birthday attack even harder, requiring 2^{80} random hashes.

SHA is very similar to MD4 [9], but has a 160-bit hash value. The main changes are the addition of an expand transformation and the addition of the previous step's output into the next step for a faster avalanche effect. Ron Rivest made public the design decisions behind MD5 [10], but SHA's designers did not. Here are Rivest's MD5 improvements to MD4 and how they compare with SHA's [6]:

1) "A fourth round has been added." SHA does this, too. However, in SHA the fourth round uses the same f function as the second round.
2) "Each step now has a unique additive constant." SHA keeps the MD4 scheme where it reuses the constants for each group of 20 rounds.
3) "The function G in round 2 was changed from $((X \wedge Y) \vee (X \wedge Z) \vee (Z \wedge Y))$ to $(X \wedge Z) \vee (Y \wedge (\neg Z)$ to make G less symmetric." SHA uses the MD4 version: $((X \wedge Y) \vee (X \wedge Z) \vee (Z \wedge Y))$.
4) "Each step now adds in the result of the previous step. This promotes a faster avalanche effect." This change has been made in SHA as well. The difference in SHA is that a fifth variable is added, and not b, c, or d, which is already used in ft. This subtle change makes the den Boer-Bosselaers attack against MD5 impossible against SHA.
5) "The order in which message sub-blocks are accessed in rounds 2 and 3 is changed, to make these patterns less alike." SHA is completely different, since it uses a cyclic error-correcting code.
6) "The left circular shift amounts in each round have been approximately optimized, to yield a faster avalanche effect. The four shifts used in each round are different from the ones used in other rounds." SHA uses a constant shift amount in each round. This shift amount is relatively prime to the word size, as in MD4.

This leads to the following comparison: SHA is MD4 with the addition of an expand transformation, an extra round, and better avalanche effect; MD5 is MD4 with improved bit hashing, an extra round, and better avalanche effect [6].

Because SHA produces a 160-bit hash, it is more resistant to brute-force attacks (including birthday attacks) than 128-bit hash functions.

IV. DSA

In August 1991, The National Institute of Standards and Technology (NIST) proposed the Digital Signature Algorithm (DSA) for use in their Digital Signature Standard (DSS). This is Public-Key Digital Signature Algorithm.

DSA is a variant of the Schnorr and ElGamal signature algorithms, and is fully described in [2]. The algorithm uses the following parameters:

p = a prime number L bits long, when L ranges from 512 to 1024 and is a multiple of 64. (In the original standard, the size of p was fixed at 512 bits [1]. This was the source of much criticism and was changed by NIST [2].)
q = a 160-bit prime factor of $p - 1$.
$g = h^{(p-1)/q} \bmod p$, where h is any number less than $p - 1$ such that $h^{(p-1)/q} \bmod p$ is greater than 1.
x = a number less than q.
$y = g^x \bmod p$.

The algorithm also makes use of a one-way hash function: $H(m)$. The standard specifies the Secure Hash Algorithm..

The first three parameters, p, q, and g, are public and can be common across a network of users. The private key is x; the public key is y.

To sign a message, m:

1) Sender generates a random number, k, less than q.
2) Sender generates

$$r = (g^k \bmod p) \bmod q$$
$$s = (k^{-1} (H(m) + xr)) \bmod q$$

The parameters r and s are her signature; she sends these to recipient.

3) Recipient verifies the signature by computing

$$w = s^{-1} \bmod q$$
$$u_1 = (H(m) \cdot w) \bmod q$$
$$u_2 = (rw) \bmod q$$
$$v = ((g^{u1} \cdot y^{u2}) \bmod p) \bmod q$$

If $v = r$, then the signature is verified.

Proofs for the mathematical relationships are found in [2].

Real-world implementations of DSA can often be speeded up through precomputations. Notice that the value r does not depend on the message. You can create a string of random k values, and then precompute r values for each of them. You can also precompute k^{-1} for each of those k values. Then, when a message comes along, you can compute s for a given r and k^{-1}. This precomputation speeds up DSA considerably.

V. DSA PRIME GENERATION

Lenstra and Haber pointed out that certain moduli are much easier to crack than others [3]. If someone forced a network to use one of these "cooked" moduli, then their signatures would

be easier to forge. This isn't a problem for two reasons: These moduli are easy to detect and they are so rare that the chances of using one when choosing a modulus randomly are almost negligible—smaller, in fact, than the chances of accidentally generating a composite number using a probabilistic prime generation routine.

In [2] NIST recommended a specific method for generating the two primes, p and q, where q divides $p - 1$. The prime p is L bits long, between 512 and 1024 bits long, in some multiple of 64 bits. The prime q is 160 bits long. Let $L - 1 = 160n + b$, where L is the length of p, and n and b are two numbers and b is less than 160.

1) Choose an arbitrary sequence of at least 160 bits and call it S. Let g be the length of S in bits.
2) Compute $U = \mathrm{SHA}(S) \oplus \mathrm{SHA}((S + 1) \bmod 2^g)$, where SHA is the Secure Hash Algorithm.
3) Form q by setting the most significant bit and the least significant bit of U to 1.
4) Check whether q is prime.
5) If q is not prime, go back to step 1.
6) Let $C = 0$ and $N = 2$.
7) For $k = 0,1,..., n$ let $V_k = \mathrm{SHA}((S + N + k) \bmod 2^g)$
8) Let W be the integer:
   $W = V_0 + 2^{160}V_1 + ... + 2^{160(n-1)}V_{n-1} + 2^{160n}(V_n \bmod 2^b)$
   and let:
   $X = W + 2^{L-1}$
   Note that X is an L-bit number.
9) Let: $p = X - ((X \bmod 2q) - 1)$. Note that p is congruent to 1 mod $2q$.
10) If $p < 2^{L-1}$, then go to step 13.
11) Check whether p is prime.
12) If p is prime, go to step 15.
13) Let $C = C + 1$ and $N = N + n + 1$.
14) If $C = 4096$, then go to step 1. Otherwise, go to step 7.
15) Save the value of S and the value of C used to generate p and q.

In [2], the variable S is called the "seed," C is called the "counter," and N the "offset".

The point of this exercise is that there is a public means of generating p and q. For all practical purposes, this method prevents cooked values of p and q. If someone hands you a p and a q, you might wonder where that person got them. However, if someone hands you a value for S and C that generated the random p and q, you can go through this routine yourself. Using a one-way hash function, SHA in the standard, prevents someone from working backwards from a p and q to generate an S and C.

This security is better than what you get with RSA. In RSA, the prime numbers are kept secret. Someone could generate a fake prime or one of a special form that makes factoring easier. Unless you know the private key, you won't know that. Here, even if you don't know a person's private key, you can confirm that p and q have been generated randomly.

VI. Security of DSA

A. Attacks against k

Each signature requires a new value of k, and that value must be chosen randomly. If anyone ever recovers k that sender used to sign a message, perhaps by exploiting some properties of the random-number generator that generated k, he can recover sender's private key, x. If anyone ever gets two messages signed using the same k, even if he doesn't know what it is, he can recover x. And with x, anyone can generate undetectable forgeries of sender's signature. In any implementation of the DSA, a good random-number generator is essential to the system's security [6].

B. Dangers from using Common Modulus

Even though the DSS does not specify a common modulus to be shared by everyone, different implementations may. For example, the Internal Revenue Service is considering using the DSS for the electronic submission of tax returns. What if they require every taxpayer in the country to use a common p and q? Even though the standard doesn't require a common modulus, such an implementation accomplishes the same thing. A common modulus too easily becomes a tempting target for cryptanalysis. It is still too early to tell much about different DSS implementations, but there is some cause for concern.

C. Subliminal Channel in DSA

Gus Simmons discovered a subliminal channel in DSA [6]. This subliminal channel allows someone to embed a secret message in his signature that can only be read by another person who knows the key. According to Simmons, it is a "remarkable coincidence" that the "apparently inherent shortcomings of subliminal channels using the ElGamal scheme can all be overcome" in the DSS, and that the DSS "provides the most hospitable setting for subliminal communications discovered to date." NIST and NSA have not commented on this subliminal channel; no one knows if they even knew about it. Since this subliminal channel allows an unscrupulous implementer of DSS to leak a piece of the private key with each signature, it is important to never use an implementation of DSS if you don't trust the implementer.

VII. Software Realization

Fig. 2 and Fig. 3 show the block diagram of the realized system for the digital signature. The program is fully written in the Delphi programming language and development environment that has been used is Borland Delphi 7. Delphi is an object-oriented programming language used to implement applications that run in the Windows operating system, and is particularly suited for graphical applications and graphical user interfaces (GUI). The syntax of the language is similar to Pascal syntax, and the original name of Delphi was Object Pascal. In addition to the main libraries that are used for programming, BigInt library is used for operations with large numbers (numbers greater than 64 bits). Procedures and

functions of BigInt library are used for measurement time, too.

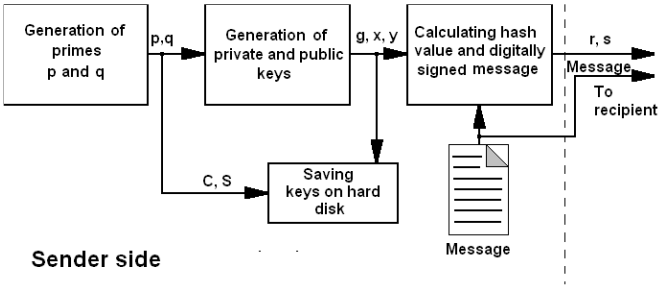

Fig. 2. Block diagram of realized system for digital signature – sender side.

Hereinafter, will be tested a presented system for digitally signing DSA, and the results will be used to compare this system with a digital signature system based on hash functions and RSA algorithm [8]. In order to credibility compare of these two systems, in both implementation used identical algorithm and the same software implementation of SHA hash functions, system simulation and measurement results were carried out on the same computer and the time test of digital signing the message used the same samples, i.e. the same file.

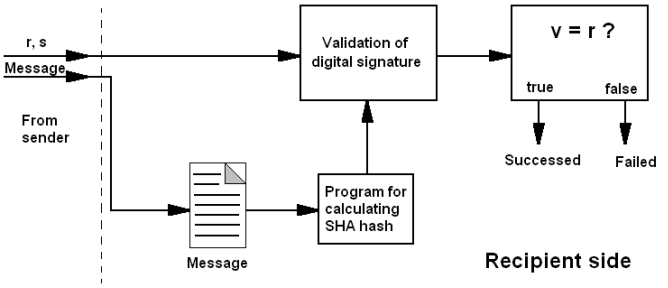

Fig. 3. Block diagram of realized system for digital signature – recipient side.

The user of software that implemented DSA may choose public and private key length by selecting the prime number p bit-length, which can be between 512 and 1024 bits, and the selected length is multiple of 64. In purpose of testing DSA algorithm were used prime numbers p of length 1024 bits, and by thus is determined the length of public keys y, which are also length 1024 bits. When you choose what will be the size of p, by pressing a button in the program generate keys and then save them in a given location on your computer. After the keys are remembered as .id files on your computer, the user is able to use a generated key many times to sign the message. The public key is freely distributed and shared with other users so that they were able to verify the signature, while the private key, or file in which is preserved, stored on a computer, thus leaving for the use only for the person whose identity is represented by the private key. Block for generating prime numbers p and q is implemented as described in Chapter 4. Due to the conditions in steps 5 and 14, mentioned in above chapter, to get the prime numbers that satisfy the conditions is necessary to repeatedly execute a block of commands, and therefore the time required to generate p and q varies depending on the number of repeat loops. Table I is given a

TABLE I
TIME TO GENERATE 1024-BIT KEYS

Time	Min [ms]	Average [ms]	Max [ms]
DSA	28.93	354.8173	1437.738
RSA	/	45.7217	/

minimum, maximum and average time needed to generate 1024-bit DSA key algorithm. By comparison, the next line in the same table gives the time needed to generate a key the same length using RSA.

For the source of messages a user can choose either a text that will write or a file from his computer (images, movies, documents ...). In case the message is chosen file, program presents that file as a sequence of bytes, and then in the sequence adds bits on way presented in the description of SHA hash function. After adding bits, the total sequence length is a multiple of 64 (one element of the array, byte = 8 bits; 64 bytes = 512 bits). When the message for signing is text entered directly in program, then each character of text will be presented to byte values corresponding to the ASCII character representation (table). On this way program makes array of bytes from text message. There is adding bits on the end of this array too, so a length is a multiple of 64. Now we compute SHA hash value of this array as described earlier in Chapter 2. By using a hash value, randomly generated number k, and previously generated and calculated parameters (p, q, g and x) is calculating a digital signature in the manner explained in the description of the DSA algorithm (Chapter 3). Times needed to compute the digital signature, depending on the size of messages are presented in Table II. These results are shown on Fig. 4, too. These sizes of test message were used because 20 KB is the average size of e-mail without attachments, 100 KB is size of a written document, 500 KB is size of medium

TABLE II
TIME [MS] NEEDED TO DIGITALLY SIGN MESSAGES OF DIFFERENT SIZES

Message	20KB	100KB	500KB	1MB	3MB
DSA	2.7197	5.3297	18.077	37.588	98.789
RSA	6.563	9.162	21.951	40.616	101,04

resolution jpg images, 1 MB is size of high resolution image, and 3 MB is size of mp3 songs.

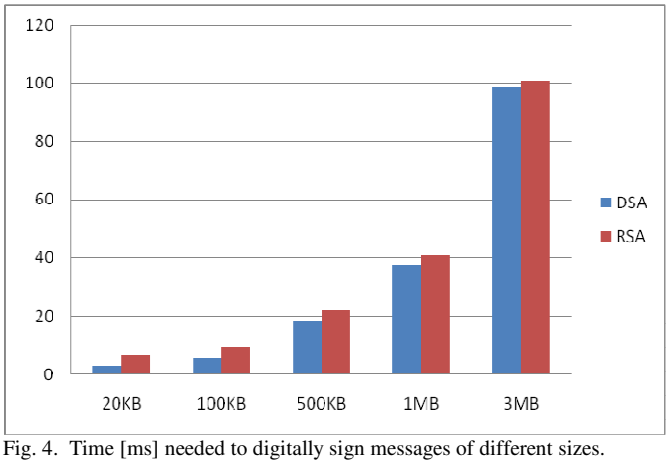

Fig. 4. Time [ms] needed to digitally sign messages of different sizes.

The validation of digital signature and sender authentication is performed in the last block of the scheme in Fig. 3. Mathematical operations, described in Chapter 3, are applied in the block for checking digital signature. These operations use the sender's public key, the received message and digital signature. First you need to calculate a hash value of message. In our test for calculating a hash value of the received message, and for checking the correctness of code that calculates the SHA hash values, we used a free SlavaSoft's HashCalc 2.02 software. At the end if v equals r then we can safely say that the message is transferred in an unmodified form and that is from the same sender whose public key we decipher the message. If the compared values are different message should be rejected because it is in transmission its content is changed or it is not from the person who allegedly signed. Signature verification procedure in the depicted software implementation of the DSA algorithm takes an average of 2.97 ms, while the same operation for the RSA algorithm required 5.817 ms.

VIII. Conclusion

With 512-bits key DSA wasn't strong enough for long-term security. With 1024-bits key it is, and for that reason is used in this software implementation.

DSA keys generation is a much slower than the RSA algorithm, the time required to digitally sign messages are similar, if larger files are signed (on the order MB and larger), while the DSA is faster at the signing of small files and text messages, as at verifying the digital signature.

Unlike RSA, DSA can't be used for the encryption or key distribution, but this is not the point of the DSS standard. This is a digital signatures standard. It is possible to use these two algorithms for realization another system for digital signatures, which would use DSA for digital signing and RSA for keys distribution. The reason for usage DSA for signing, and not RSA, is security, because it offers the possibility to check the manners in which primes p and q are generated. The RSA algorithm, prime numbers are kept confidential, and verification that they intentionally used their specific values is possible only with the knowledge of the private key. At DSA algorithm only publicly available variables S and C (seed and counter) can be used for checking without disclosing the private key.

References

[1] National Institute of Standards and Technology, NIST FIPS PUB XX. „Digital Signature Standard“, U.S. Departmentof Commerce, DRAFT, 19 Aug 1991.

[2] National Institute of Standards and Technology, NIST FIPS PUB 186. „Digital Signature Standard“, U.S. Departmentof Commerce, May 1994.

[3] A. K. Lenstra and S. Haber, letter to NIST Regarding DSS, 26 Nov 1991.

[4] G. J. Simmons, „The Subliminal Channels of the U.S. Digital Signature Algorithm (DSA)“, Proceedings of the Third Symposium on: State and Progress of Research in Cryptography, Rome: Fondazone Ugo Bordoni, 1993, pp. 35 – 54.

[5] D. Drajić, P. Ivaniš, “Uvod u teoriju informacija i kodovanje”, *Akademska misao*, Beograd, 2009.

[6] B. Schneier, “Primenjena kriptografija”, *Mikro knjiga*, Beograd, 2007.

[7] IETF RFC 4634 - US Secure Hash Algorithms (SHA and HMAC-SHA), http://www.ietf.org/rfc/rfc4634.txt.

[8] B. Pajčin, P Ivaniš, “Softverska realizacija sistema za digitalno potpisivanje sa heš funkcijama i RSA algoritmom”, *Infoteh-Jahorina 2011*, Jahorina 2011.

[9] IETF RFC 1186 - The MD4 Message Digest Algorithm, http://www.ietf.org/rfc/rfc1186.txt.

[10] IETF RFC 1321 - MD5 Message-Digest Algorithm, http://www.ietf.org/rfc/rfc1321.txt.