Can D.S.A. be Improved ?
- Complexity Trade-Offs with the Digital Signature Standard -

David NACCACHE[1]★, David M'RAÏHI[1], Serge VAUDENAY[2] and Dan RAPHAELI[3]

[1] GEMPLUS CARD INTERNATIONAL, Crypto Team, 1 place de Navarre, F-95208, Sarcelles CEDEX, FRANCE
`{100142.3240 and 100145.2261}@compuserve.com`

[2] ECOLE NORMALE SUPERIEURE, G.R.E.C.C. - D.M.I., 45 rue d'Ulm, F-75230, Paris CEDEX 5, FRANCE
`serge.vaudenay@ens.fr`

[3] CALIFORNIA INSTITUTE OF TECHNOLOGY, Electrical Engineering, Pasadena, CA 91103, USA
`dani@romeo.caltech.edu`

Abstract : The Digital Signature Algorithm (DSA) was proposed in 1991 by the US National Institute of Standards and Technology to provide an appropriate core for applications requiring digital signatures. Undoubtedly, many applications will include this standard in the future and thus, the foreseen domination of DSA as a legal certification tool is sufficiently important to focus research endeavours on the suitability of this scheme to various situations.

In this paper, we present six new DSA-based protocols for :

• Performing a quick batch-verification of n signatures.
The proposed scheme allows to make the economy of $\approx 450n$ modular multiplications.
• Avoiding the cumbersome calculation of $1 / k \bmod q$ by the signer.
• Compressing sets of DSA transactions into shorter *archive signatures.*
• Generating signatures from pre-calculated "Use & Throw" 224-bit *signature-coupons.*
• Self-certifying the moduli and bit-patterning directly q on p (gain of 60.4% in key size).

All our schemes combine in a natural way full DSA compatibility and flexible trade-offs between computational complexity, transmission overheads and key sizes.

1. Introduction

There is no doubt that the proposal of the DSA [1] as a *Federal Standard* will make this scheme widely accepted and used for certifying the integrity of messages and documents. Electronic passports, ID-cards, driving licenses and other DSA-based purses are thus expected to join (and partially replace) a whole gamut of passive devices used today in telephony and banking.

However, many potential applications will certainly remain beyond of reach because of the difficulty of manipulating big-numbers in portable devices and although constant improvements of silicon technologies may solve the majority of these problems, the price of crypto-dedicated chips will most certainly be the new barrier to jump over during the next decade.

Nevertheless, DSA is aimed to be only one layer in more complex systems which will generally include phone lines, powerful (but potentially hostile) terminals and storage or compression facilities which might be of some help in the processing of DSA signatures.

This paper presents a family of DSA-compatible protocols precisely designed to simplify the signer and the verifier's work whenever possible. For instance, one of our methods saves 450 multiplications per signature by batch-processing simultaneously large sets of signatures, a second algorithm compresses sets of DSA transactions into shorter *archive signatures* and two other protocols avoid the computations of modular inverses and exponentials to the signer.

All these schemes combine in a natural way full DSA compatibility and flexible trade-offs between computational complexity, transmission overheads and key sizes.

★ This work was started while visiting J.P.L. and CalTech's Electrical Engineering Department in the summer of 1993.

2. DSA batch verification

The parallel verification of many signatures in a single algorithmic operation might be of big help in many practical applications : banks will collect sequentially considerable volumes of transactions but compensate them at once, toll-highway machines will process collectively many electronic tokens at rush-hour peaks and pay-TV operators might be interested in satisfying large numbers of customers in a relatively short time.

The algorithm presented in this section is *provably* as secure as the original DSA but much faster as it replaces multiplications modulo p by additions modulo q.

For simplicity, the protocol will be described for a single signer as their extension to a community of users (same public moduli but different public-keys) is straightforward but requires heavier notations :

for $i = 1$ to n, the signer :

① picks $k_i \in_R GF^*(q)$, computes $\lambda_i = g^{k_i} \bmod p$, $s_i = \dfrac{\mathrm{SHA}(m_i) + x\,\lambda_i}{k_i} \bmod q$,

② and sends $\{\lambda_i, s_i, m_i\}$ to the verifier.

To verify $\{\lambda_i, s_i, m_i\}_{i=1,\dots,n}$:

① pick n pairwise relatively prime *equation randomisers* $b_1,\dots,b_n \in_R GF^*(q)$,

② test if: $\prod_{i=1}^{n} \lambda_i^{b_i} \bmod p \equiv g^{\sum_{i=1}^{n} b_i w_i \,\mathrm{SHA}(m_i) \bmod q}\; y^{\sum_{i=1}^{n} b_i w_i \lambda_i \bmod q} \bmod p$

③ and replace $\{\lambda_i, s_i, m_i\}_{i=1,\dots,n}$ by $\{r_i = \lambda_i \bmod q, s_i, m_i\}_{i=1,\dots,n}$

The security of this scheme is guaranteed by :

Theorem 1 *The following statements are equivalent :*

① *There is an efficient algorithm* $\mathbb{C}(m_1, m_2, p, q, g, y, b, b_1, b_2) = \{s_1, s_2, \lambda_1, \lambda_2\}$ *such that :*

$$\lambda_1^b \lambda_2^{b_i} \equiv g^{bw_1 \mathrm{SHA}(m_1) + b_i w_2 \mathrm{SHA}(m_2) \bmod q}\, y^{bw_1\lambda_1 + b_i w_2 \lambda_2 \bmod q} \bmod p \textit{ for } i = 1,2 \textit{ and } b_1 \neq b_2$$

② *There is an efficient algorithm which breaks DSA.*

Proof: ②⇒① is straightforward. To prove that ①⇒②, pick any b, choose $\{b_1, b_2\}$ such that $\tilde{b} = b_1 - b_2$ has an inverse modulo q and compute $\mathbb{C}(m_1, m_2, p, q, g, y, b, b_1, b_2) = \{s_1, s_2, \lambda_1, \lambda_2\}$. Dividing the formulae :

$$\lambda_1^b \lambda_2^{b_i} \equiv g^{bw_1 \mathrm{SHA}(m_1) + b_i w_2 \mathrm{SHA}(m_2) \bmod q}\, y^{bw_1\lambda_1 + b_i w_2 \lambda_2 \bmod q} \bmod p \quad \text{for } i = 1,2$$

we get : $\lambda_2^{\tilde{b}} \equiv g^{\tilde{b} w_2 \mathrm{SHA}(m_2) \bmod q}\, y^{\tilde{b} w_2 \lambda_2 \bmod q} \bmod p$ and $\tilde{b}$ th roots mod p can be taken at both sides (as $\tilde{b}$ has an inverse modulo q) to see that $\{m_2, s_2, r_2 = \lambda_2 \bmod q\}$ passes the <u>sequential</u> DSA test. □

An immediate consequence of theorem 1 (which can be further generalised) is :

Lemma 1 *If the* b_i *s are pairwise relatively prime, strictly smaller than* q *and picked in a set* A *then :*

$$\Omega(A) = \Pr\left[BATCH\left(b_i \in A, \{\{r,s\},\{r',s'\}\}\right) = True \;\middle|\; DSA(\{r,s\}) \wedge DSA(\{r',s'\}) = False\right] = \frac{1}{|A|^2 - |A|} \cong |A|^{-2}$$

The most natural A is the set of primes smaller than a certain bound (for instance , $\Omega(\{\text{primes} \le 7927\}) \cong 10^{-6}$ is suitable for most applications), but the cardinality of A can be optimised by the following technique :

Denote $C = \{c \text{ first primes}\}$ and let $f(c)$ be such that $S_c^{f(c)}$ is maximal[1]. To compute $b_1,\ldots,b_{f(c)}$, generate a random partition of C to $f(c)$ classes $\{C_1,\ldots,C_{f(c)}\}$ and inter-multiply the elements in each C_i to form b_i.

The values $c = 54$ or 6542 are particularly interesting as they correspond to the primes smaller than 2^8 and 2^{16} which can be easily manipulated by computers.

A couple of noteworthy results is :

Lemma 2 *If all the C_is have exactly the same cardinality d, then* $\Omega(C) = \dfrac{d!^2 (c-2d)!}{c!}$

Lemma 3 *If the b_is are chosen freely in* $\{0,1\}^e$ *then* $\Omega\left(\{0,1\}^e\right) = \dfrac{1}{2^e}$

and $f(c)$ can be easily approximated by **Harper's lemma** [7] :

Lemma 4 $\lim\limits_{c \to \infty} \dfrac{c}{f(c)\ln(c)} = \bar{e}$ (where $\bar{e} = 2.71828\ldots$ denotes the base of natural logarithms).

Characteristic instances of the two verification strategies (sequential and batch for lemma 3 and $e = 20$) are compared in table 1 where all costs are given for n signatures.

scheme ⇨ *trade-off ⇩*	*NIST-DSA* *sequential verification*	*Probabilistic* *batch verification*
# of 160 bit mult.	$2\,n$	$4\,n$
# of 512 bit mult.	$\approx 475\,n$	$\approx 29n + 474$
size of e		20 bits
transmission	40 n bytes	84 n bytes
DSA format	by definition	with $r = \lambda \mod q$

Table 1 : DSA verification strategies

3. Compressing DSA signatures

DSA signatures are meant to be legal proofs and should thus be archived during a certain time. Taking as an everyday example the case of cash dispensing machines, each money withdrawal operation makes the terminal generate a short *debit certificate* (32 bytes) which is kept for future trace by the bank during four years. Although public key techniques are ideally tailored for such situations (attacks against terminals are of no help for forging cards), it is striking to observe that the DSA "protection layer" may turn to be as big as the protected message itself.

Whilst messages may contain redundancies and admit a whole gamut of compression solutions, DSA signatures (based on k and x, both of which are random and unknown to the archivist) are, *à priori*, impossible to compress since $\log(x) + \log(k) \approx \log(s) + \log(r)$.

Coming back to our bank example, and assuming that the card-holder comes to the bank office from time-to-time (even once a year), one can imagine a *signer-aided compression scheme* in which the bank will take advantage of the visits to re-send to the card a set of signatures (of which the signer kept no trace), the card will verify these and once recognised (as generated with its x), concatenate all the corresponding messages and put a single signature on the whole. Unfortunately, this solution suffers from the heavy disadvantage of forcing the card to perform a number of DSA verifications which might be too lengthy and unsuited to a real-life context.

[1] S_n^m denotes the Stirling numbers of the 2nd kind (that is, the # of partitions of a set of n objects to m classes).

However, the card knows more than the verifier about its own signatures, namely, knowing x, the signer can easily reverse the steam and compute k. Thus, by a proper redundancy in k, the card can recognise its signatures with a minimal effort (a couple of multiplications) .

Practically, we propose to build : $k = k' \left| \left(\mathrm{SHA}(k' | \mathrm{SHA}(m)) \bmod 2^{\lfloor size(q)/3 \rfloor} \right) \right.$ where $k' \le \left\lfloor \sqrt[3]{q^2} \right\rfloor$ is a shorter random picked by the signer during the generation of the signature and m the signed message.

The compression protocol (now being implemented in an Asian banking application) is :

① The signer sets $h_0 = \varnothing$

② for $i = 1$ to n :

- The archivist sends to the signer the triple $\left\{ h_i = \mathrm{SHA}(m_i),\ w_i = \frac{1}{s_i} \bmod q,\ r_i \right\}$
- The signer checks if $\exists\ \ell \le \left\lfloor \sqrt[3]{q^2} \right\rfloor$ such that $\ell \left| \left(\mathrm{SHA}(\ell \,|\, h_i) \bmod 2^{\lfloor size(q)/3 \rfloor} \right) \right. = w_i (h_i + x\, r_i) \bmod q$

  and updates $h_0 = \mathrm{SHA}(h_i | h_0)$ if this test succeeds and $r_i \neq 0$

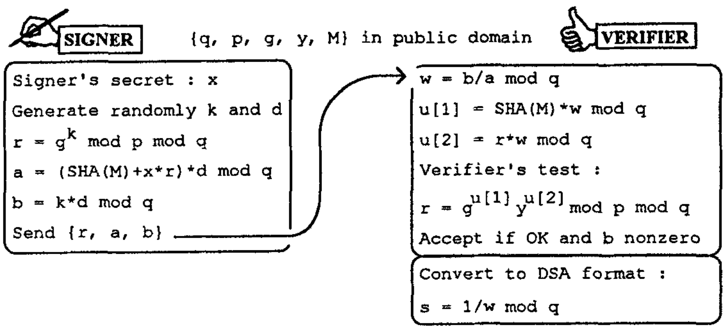
③ The signer signs h_0 and the archivist checks and files this signature before erasing $\{s_i, r_i\}$ for $i = 1, \ldots, n$.

Note that the 2/3-sized random was chosen to reach an optimal birthday-paradox security ($\approx 2^{53}$ for $q \approx 2^{160}$).

4. 1/*k*-less signatures

For generating or verifying DSA signatures, one must posses an algorithmic tool for computing modular inverses. The NIST recommends Euclid's algorithm which is fast and suitable to software applications but becomes of no interest when the basic tool at one's command is a ready-to-use modular multiplier ("back-box" implementing the operation $a \times b \bmod n$).

This observation may well explain the fact all today's DSA smart-card prototypes (three different crypto-engines and five different companies[2] !) use the identity : $x^{-1} = x^{q-2} \bmod q$ to compute modular inverses[3].
In this section we propose a simpler solution which eliminates completely $1/k$ from the signature generation process. The idea consists in hiding k behind a random *blinder* d which disappears automatically during the verification. The proposed protocol is :



```
Signer's secret : x
Generate randomly k and d
r = g^k mod p mod q
a = (SHA(M)+x*r)*d mod q
b = k*d mod q
Send {r, a, b}
```

```
w = b/a mod q
u[1] = SHA(M)*w mod q
u[2] = r*w mod q
Verifier's test :
r = g^u[1] y^u[2] mod p mod q
Accept if OK and b nonzero

Convert to DSA format :
s = 1/w mod q
```

[2] Fortress-Thomson on ST16CF54, Gemplus-CCETT on 83C852 and Siemens on SLE 44C200.

[3] This method yields a ratio of $\left(\frac{\log(q)}{\log(p)}\right)^2$ ($\approx$ 10% for a 512-bit p) between the computation times of $1/k$ and r.

and its security is guaranteed by the fact that any would-be cryptanlysis $\mathfrak{C}(m,p,q,g,y)=\{r,a,b\}$ will break the original DSA by post-calculating $s = a / b \bmod q$.

5. Shorter self-certifying primes

In memory-restricted environments (like smart-cards or other portable electronic tokens), key size is an influential parameter in the favour of a given algorithmic solution. The standard specifies a "wild" prime generation scheme (meant to avoid trapdoor moduli) which outputs at least 844 bits : p and q (respectively 512 and 160 bits long) and a *certificate of proper prime generation* (172 bits at minimum).

Our approach for reducing the size of these data (by about 60%) combines several advantages :

① q is simply the 160 most significant bits of p.
② The certificate of proper prime generation is embedded into p as well.
③ The "wildness" of our prime generation procedure still avoids the generation of trapdoor primes.

The algorithm is :

Steps 1 to 4. (See appendix 2) Identical to the NIST key generation algorithm with a 160-bit *Seed*.
Step 5. $p = q \mid Seed \mid 32 \text{ zero bits} \mid \text{SHA}(Seed)$
Step 6. $p = p - (p \bmod q) + 1$
Step 7. $p = p + q$
Step 8. If the 32-bit zero pattern in p (*counter*) turned to $7FFFFFFF_{16}$ go to **Step 1.**
Step 9. If p is composite go to Step 7.

and an output example (*Seed* and q are taken from the NIST's DSA example-list) is :

```
q       = b20db0b101df0c6624fc1392ba55f77d577481e5
Seed    = d5014e4b60ef2ba8b6211b4062ba3224e0427dbd
counter = 0000000c
"tail"  = fdb18bdb74205335fa5302b67a7db7c08a12ad41
```

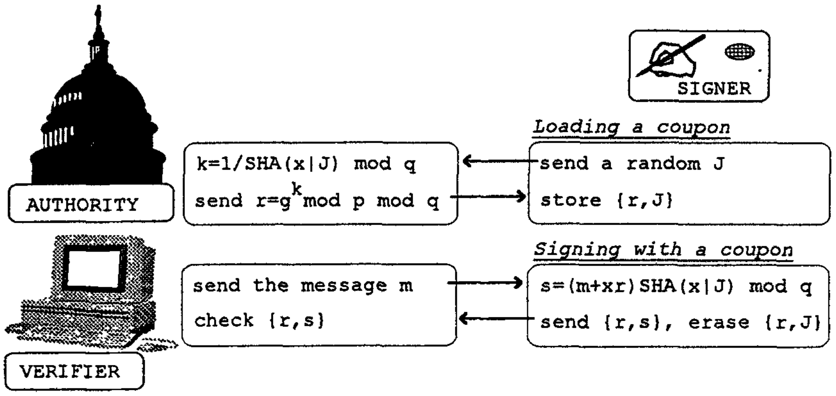
which concatenation gives :

```
p       = b20db0b101df0c6624fc1392ba55f77d577481e5|d5014e4b60ef2ba8b6211b40
         62ba3224e0427dbd|0000000c|fdb18bdb74205335fa5302b67a7db7c08a12ad41
```

6. Use & Throw DSA signatures

A well-known feature of the DSA, inherited from its ancestors **El-Gamal** [5] and **Schnorr** [10], is the possibility to pre-compute r and the inverse of k before the message is known. Then, the effort needed to produce s from m is negligible. This section introduces a coupon-based protocol exploiting this property for helping the signer to generate signatures very quickly. In our model, a trusted authority sends public data packets (*Use & Throw coupons*) to the signer who stores them for future use. Each coupon is *only* 28-byte long and enables its owner to generate one DSA signature (if a coupon is used twice, the signer's x is revealed). Two noteworthy advantages of this method are that the signer has only to posses x and q (the storage of g and p, which represents 1024 bits at minimum, can be avoided) and only a couple of multiplications is needed to transform a coupon to a signature.

The system is based on a retro-calculation of k from an easily compressible inverse and is ideally suited to electronic-purse applications where card-holders interact periodically with a trusted authority for loading money into their purses (refreshing the coupon's reserve) :



This scheme was implemented on a 68HC05-based prototype[4] which generates s in less than 150 ms (4 MHz clock) and can contain up to 91 coupons in EEPROM. A *heavy-duty* version (now under development) will be 30% faster and tailored to contain 279 coupons.

Note that when **Montgomery's algorithm** [8] is used (let $Q = 2^{-size(q)} \bmod q$), the signer can shortcut his calculations by using the key : $x' = xQ^{-1} \bmod q$ if the authority compensates :

$$r = \left(\sqrt[Q]{g} \bmod p \right)^k \bmod p \bmod q$$

The coupon-owner will then compute s by two **Montgomery** rounds (instead of four) :

① $z = Montgomery_q(x', r) \equiv x'rQ \equiv xQ^{-1}rQ \equiv xr \bmod q$

② $s = Montgomery_q(z+m, \mathrm{SHA}(J|x)) \equiv (z+m)\mathrm{SHA}(J|x)Q \bmod q$

☞ The signature is still DSA-compatible and the storage of $4^{size(q)} \bmod q$ (20 bytes normally needed for converting results from Montgomery's format to the conventional number system) has been avoided.

☞ Note that coupons can be reduced to exactly 20-bytes if only one new common J is generated during each loading session and inverses are diversified by $1/k_i = \mathrm{SHA}(J|\ x|\ i)$ where i is the coupon's number.

The general electronic check concept (see for instance [4]) can apply more or less efficiently (size of the check and the number of multiplications required to produce a signature) to a big variety of cryptosystems. We incite the reader to examine and/or modify the relevant bibliography for obtaining an optimal suitability to a given practical situation.

7. Conclusion

We showed that relatively simple considerations may greatly accelerate the processing of DSA signatures. The main characteristics of the algorithms presented in this article (which can be easily adapted to suit different environnements or even other signature schemes like the **Brickell-McCurley** [3] and **Guillou-Quisquater** [6]) are summarised in table 2 where the first column indicates the NIST's method for reference.

[4] ST16623 (no crypto-engine aboard).

scheme ⇨ ***trade-off*** ⇩	***NIST-DSA (signer)***	***1/k-less DSA (signer)***	***U&T-DSA (signer)***	***DSA-compress (n signatures)***	***auto-certified p (both sides)***
# of 160 bit mult.	2	3	2	$2(n+1)$	unchanged
# of 512 bit mult.	≈ 237	≈ 237	none	≈ 237	unchanged
modular inverse	yes	no	no	yes	unchanged
transmission	40 bytes	60 bytes	68 bytes	$40(n+1)$ bytes	unchanged
size of moduli	84 bytes	84 bytes	20 bytes	84 bytes	64 bytes
DSA format	by definition	with $s = a/b \bmod q$	yes	yes	partial
patents by	N.I.S.T.	Gemplus Card International			public domain

Table 2 : DSA signature generation methods

Acknowledgements

We thank **Beni Arazi, Josh Benaloh, David Chaum, Jacques Stern** and **Jean-Pierre Tillich** for their pertinent suggestions concerning this work.

References

[1] FIPS PUB XX, February 1, 1993, *Digital Signature Standard.*

[2] **E. Brickell, D. Gordon** and **K. McCurley,** *Fast exponentiation with precomputation,* technical report no. SAND91-1836C, Sandia National Laboratories, Albuquerque, New-Mexico, October 1991.

[3] **E. Brickell** and **K. McCurley,** *An interactive identification scheme based on discrete logarithms and factoring,* Journal of Cryptology, vol 5, no. 1, 1992.

[4] **D. Chaum** and **J. Bos,** *Smart Cash: A practical electronic payment system,* CWI-Report CS-R9035, August 1990.

[5] **T. El-Gamal,** *A public-key cryptosystem and a signature scheme based on discrete logarithms,* IEEE TIT, vol. IT-31:4, pp 469-472, 1985.

[6] **L. Guillou** and **J.J. Quisquater,** *A practical zero-knowledge protocol fitted to security microprocessor minimising both transmission and memory,* Advances in cryptology: Proceedings of Eurocrypt'88, LNCS, Springer-Verlag, Berlin, 330, pp 123-128, 1988.

[7] **L.H. Harper,** *Stirling behavior is asymptotically normal,* Annals of Mathematical Statistics, vol. 38, pp. 410-414, 1967.

[8] **P. Montgomery,** *Modular multiplication without trial division,* Mathematics of Computation., vol. 44(170), pp. 519-521, 1985.

[9] **J.H. Morris,** *Lambda-calculus models of programming languages,* Ph.D. thesis, MIT, 1968.

[10] **C. Schnorr,** *Efficient identification and signatures for smart-cards,* Advances in cryptology: Proceedings of Eurocrypt'89 (G. Brassard ed.), LNCS, Springer-Verlag, Berlin, 435 (1990), pp. 239-252.

Appendix 1
The Digital Signature Algorithm

The Digital Signature Algorithm (DSA), proposed in August 1991 by the US National Institute of Standards and Technology, is a DLP-based cryptosystem which parameters are :

① A prime modulus p where $2^{L-1} < p < 2^{L}$ for $512 \le L \le 1024$ and $L \mod 64 = 0$.

② A prime q such that $2^{159} < q < 2^{160}$ and p-1 is a multiple of q.

③ A number $g = h^{(p-1)/q} \mod p$ for some h.

④ A 160-bit secret-key x and an L-bit public-key y defined by the relation : $y = g^{x} \mod p$.

The integers p, q and g are system parameters and can be public and/or common to a group of users.
A 160-bit random k, used by the signer, must be kept secret and regenerated for each signature.

In order to sign a message m (hashed value of a primitive file M), the signer computes the signature $\{r, s\}$ by :

$$r = \left(g^{k} \mod p\right) \mod q \quad \text{and} \quad s = \frac{m + xr}{k} \mod q$$

To check $\{r, s\}$, the verifier computes :

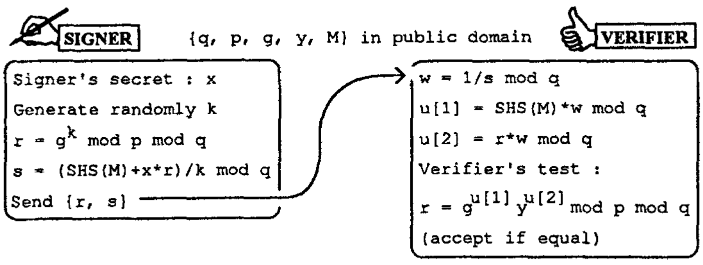
$$w = \frac{1}{s} \mod q, \qquad u_1 = m\,w \mod q \qquad \text{and} \qquad u_2 = r\,w \mod q$$

And compares if $r == \left(g^{u_1} y^{u_2} \mod p\right) \mod q$ to accept or reject the signature

Assuming no algorithmic sophistications†, the resources necessary for the implementation of the DSA are :

resources ⇩	*Signer*	*Verifier*
# of 160 bit mult.	2	2
# of 512 bit mult.	≈ 237	≈ 475
modular inverse	yes	yes
transmission	40 bytes	
size of moduli	84 bytes	

And the complete process is briefly summarised by the following figure :



```
SIGNER                {q, p, g, y, M} in public domain            VERIFIER

Signer's secret : x                          w = 1/s mod q
Generate randomly k                          u[1] = SHS(M)*w mod q
r = g^k mod p mod q                          u[2] = r*w mod q
s = (SHS(M)+x*r)/k mod q                     Verifier's test :
Send {r, s}                                  r = g^u[1] y^u[2] mod p mod q
                                             (accept if equal)
```

Figure 3. The NIST's Digital Signature Algorithm

† Some of which [2] may spectacularly divide all the 512-bit figures by about 6 but these tools apply exactly in the same manner to our schemes. The important point is the *ratio* between the protocols which remains constant whatever exponentiation strategy is used.

Appendix 2
The DSA Prime Selection Scheme

The NIST suggests to generate p and q with the following algorithm :

Step 1. Choose an arbitrary sequence of g (≥ 160) bits and call it *Seed.*

Step 2. Set : $q = \left(\mathrm{SHA}(Seed) \oplus \mathrm{SHA}(Seed + 1 \bmod 2^g)\right) \vee \left(2^{L-1} + 1\right)$

Step 3. Use a robust[†] primality testing algorithm to test whether q is prime.

Step 4. If q composite, go to Step 1

Step 5. Let *counter* = 0 and *offset* = 2

Step 6. For $k = 0,\ldots,n$ let $V_k = \mathrm{SHA}(Seed + offset + k \bmod 2^g)$

Step 7. Let : $W = V_0 + V_1\, 2^{160} + \ldots + V_{n-1}\, 2^{160(n-1)} + V_n\, 2^{160n}$ and $X = W + 2^{L-1}$ (see ‡)

Step 8. Set : $p = X - (X \bmod 2q - 1)$ to make p congruent to 1 modulo $2q$.

Step 9. If $p < 2^{L-1}$ go to Step 12

Step 10. Perform a robust primality test on p.

Step 11. If p passes the test in Step 10 go to Step 14

Step 12. Let *counter* = *counter* +1 and *offset* = *offset* + n +1

Step 13. If *counter* $\geq$ 4096 go to Step 1, otherwise go to Step 6

Step 14. Save the value of *Seed* and the value of *counter* for use in certifying the proper generation of p and q.

† A robust primality test algorithm is one where the probability of a non-prime passing the test is $< 2^{-80}$
‡ $1 < W < 2^{L-1}$ and hence $2^{L-1} \leq X < 2^L$

Appendix 3
A Remark Concerning the SHA Subfunctions f_1 and f_3

Most microprocessors perform arithmetic operations in a special memory register called *accumulator* (A).
In this model, most operations can be looked upon as belonging to two groups : *move operations* (copy of information from one memory location to another) and *arithmetic operations* of the form A ← A operator *data.*
Thus, the SHA subfunction $f_1(x,y,z) = (x \wedge y) \vee (\neg x \wedge z)$ will be evaluated by the sequence :

```
① A    ← x           ④ A   ← x        ⑦ A   ← A ∨ temp
② A    ← A ^ y       ⑤ A   ← ~A
③ temp ← A           ⑥ A   ← A ^ z
```

A well-known technique, borrowed from lambda calculus [7], allows to optimise the number of data moves by using an equivalent binary expression having a minimal number of leaves.
For f_1, one can use the identity : $z \oplus x \wedge (y \oplus z) = (x \wedge y) \vee (\neg x \wedge z)$ which compiles to :

```
① A    ← y           ③ A   ← A ^ x
② A    ← A ⊕ z       ④ A   ← A ⊕ z
```

and needs no temporary variables (this normal form is *provably optimal*).

Similarly, $f_3 = (x \wedge y) \vee (x \wedge z) \vee (y \wedge z)$ yields[5] the expression $x \wedge (y \oplus z) \oplus (z \wedge y)$ and the two new formulae speed the hashing time by ≈ 2.6% (assembly on DSP 56000).

[5] it is possible to *prove* that the evaluation of f_3 will always require *at least* one temporary variable