A Background

In the appendix different protocol is presented an is according to [25, 34]. The complexity to some of these protocols is estimated based on O-notation.

A.1 Challenge responds protocol

A.1.1 Symmetric

Unilateral authentication that is timestamp based

Algorithm 13 Unilateral authentication with timestamp

1: $A \longrightarrow B: \quad E_K(t_A, B^*)$

(*)optional data field.

Unilateral authentication with random number

Algorithm 14 Unilateral authentication with random number

1: $B \longrightarrow A: \quad r_B$

(*) optional data field.

A.1.2 Mutual authentication based on one way function

Algorithm 15 Mutual authentication based on one way function

1: $B \longrightarrow A: \quad r_B$
2: $A \longrightarrow B: \quad r_A, h_K(r_A, r_B, B)$
3: $B \longrightarrow A: \quad h_K(r_B, r_A, A)$

A.1.3 Public-key techniques

Identification based on public-key encryption

An identification protocol based on public-key encryption is presented in algorithm 16, according to [25].

Algorithm 16 Identification based on public-key encryption

1: $B \longrightarrow A: \quad h(r), B, P_A(r, B)$
2: $A \longrightarrow B: \quad r$

Modified Needham-Schroeder Public-key protocol

The Modified Needham-Schroeder Public-key protocol is presented in algorithm 17, according to [25].

Algorithm 17 Modified Needham-Schroeder Public-key protocol

1: $A \longrightarrow B: \quad P_B(r_1, A)$
2: $B \longrightarrow A: \quad P_A(r_1, r_2)$
3: $A \longrightarrow B: \quad P_B(r_2)$

A.2 Zero-knowledge protocol

A.2.1 GQ identification protocol

GQ identification protocol is presented in algorithm 18, and the complexity in table 23. In table 23 v' and n' represent the number of bit in v and n (which is equal to $(x' = \mathrm{Log}_2\, x)$).

In this protocol A prove its identity by knowledge of S_A. T-trusted party. Use RSA $n = pq$, n is public, p and q is secret. T define the public exponent $v \geq 3$, where $\gcd(v, \phi) = 1$, where $\phi) = (p-1)(q-1)$, and compute its private exponent $s = v^{-1} \mod \phi$ (v, n) is made available. A is given a unique identity I_A, $J_A = f(I_A)$ where $1 < J_A < n$, where $\gcd(J_A, \phi) = 1$. T gives to A the secret $S_A = J_A^{-s} \mod n$. A select a random secret integer r, $1 \leq r \leq n-1$ and compute $x = r^v \mod n$, B select a random integer e where $1 \leq e \leq v$, A compute $y = rS_A^e \mod n$, B receive y, construct J_A from I_A using f and compute $z = J_A^e y^v \mod n$ according to equation A.1, and accept identity if $z = x$ and $z \neq 0$.

$$\begin{aligned} J_A^e y^v \bmod n &= J_A^e (rS_A^e)^v \bmod n \\ &= J_A^e J_A^{-(sev)} \bmod n = J_A^e J_A^{-(v^{-1}ev)} \bmod n = J_A^e J_A^{-e} \bmod n = x, \end{aligned} \tag{A.1}$$

Algorithm 18 GQ identification protocol

1: $A \longrightarrow B:$ $I_A, x = r^v \bmod n$
2: $B \longrightarrow A:$ $e; (1 \leq e \leq v)$
3: $A \longrightarrow B:$ $y = rS_A^e \bmod n$

Equation	*Computation complexity* $(x' = \mathrm{Log}_2\, x)$	*Min/Max*	*Min*	*Max*
$I_A, x = r^v \bmod n$	$O(v' \cdot n'^2)$	$3 \leq v < n$	$O(n'^2)$	$O(n'^3)$
$e; (1 \leq e \leq v)$	-	-	-	-
$y = rS_A^e \bmod n$	$O(e' \cdot n'^2 + n'^2)$	$1 \leq e < n$	$O(n'^2)$	$O(n'^3)$
$z = J_A^e y^v \bmod n$	$O(e' \cdot n'^2 + v' \cdot n'^2 + n'^2)$	$3 \leq v < n$ $1 \leq e < n$	$O(n'^2)$	$O(n'^3)$
Initiator			$O(n'^2)$	$O(n'^3)$
Verifier			$O(n'^2)$	$O(n'^3)$

Table 23: Computation complexity in GQ identification protocol

A.2.2 Schnorr identification protocol

Schnorr identification protocol is presented in algorithm 19, and the complexity in table 24. In table 24 r', q' and p' represent the number of bit in r, q and p (which is equal to $(x' = \mathrm{Log}_2\, x)$).

In this protocol A proves its identity to B in a 3-pass protocol. Selection of system parameters: A suitable prime p is selected such that $p - 1$ is divisible by another prime

q. Discrete logarithms modulo p must be computationally infeasible. It is recommended that $p \approx 2^{1024}$ and $q \geq 2^{160}$.An element β is chosen, $1 \leq \beta \leq p-1$, having multiplicative order q. Each party obtains an authentic copy of the system parameters (p, q, β) and the verification function (public key) of the trusted party T, allowing verification of T's signatures $S_T(m)$ on messages m. (S_T involves a suitable known hash function prior to signing, and may be any signature mechanism.) A parameter t (e.g. $t \geq 40$), $2^t < q$, is chosen (defining a security level 2^t). Selection of per-user parameters: Each claimant A is given a unique identity I_A. A chooses a private key a, $0 \leq a \leq q-1$, and computes $v = \beta^{-a} \bmod p$. A identifies itself by conventional means (e.g., passport) to T , transfers v to T with integrity, and obtains a certificate $cert_A = (I_A, v, S_T(I_A, v)))$ from T binding I_A with v.

Algorithm 19 Schnorr identification protocol

1: $A \rightarrow B : cert_A, x_A = \beta^{r_A} \bmod p$, where $1 \leq r_A \leq q-1$
2: $B \rightarrow A : e_B$, where $1 \leq e_B \leq 2^t < q$
3: $A \rightarrow B : y_A = a_A e_B + r_A \bmod q$
4: B Verify that $z = \beta^y v^e \bmod p$ and that $z = x_A$

According to [25] the size of p should be 1024 bit, and q at least 160 bit. This give the same security as 1024 RSA, according to table 36 and 37.

Equation	*Computation complexity* $(x' = \text{Log}_2\, x)$	*Min/Max*	*Min*	*Max*
$x_A = \beta^{r_A} \bmod p$	$O(r'_A p'^2)$	$1 \leq r_A \leq q$	$O(p'^2)$	$O(q' \cdot p'^2)$
e_B	-	-	-	-
$y_A = a_A e_B + r_A \bmod q$	$O(q'^2 + q)$	$1 \leq e_B \leq q$	$O(q'^2)$	$O(q'^2)$
$z = \beta_A^y v^{e_B} \bmod p$	$O(y'_A p'^2 + e'_B p'^2 + p')$	$y_A = q$ $1 \leq e_B \leq q$	$O(q' \cdot p'^2)$	$O(q' \cdot p'^2)$
Initiator			$O(p'^2)$	$O(q' \cdot p'^2)$
Verifier			$O(q' \cdot p'^2)$	$O(q' \cdot p'^2)$

Table 24: Computation complexity in Schnorr identification protocol

A.3 Digital Signatures

A.3.1 RSA signature scheme

The RSA signature scheme is presented in algorithm 20, and the complexity in table 25. In table 25 d', e' and n' represent the number of bit in d, e and n (which is equal to $(x' = \mathrm{Log}_2\, x)$). Each entity creates an RSA public key and a corresponding private key, according to 20.

Algorithm 20 RSA signature scheme

Key generation for RSA:
Generate two large distinct random numbers primes p and q, each rougly the same size. Compute $n = pq$ and $\phi = (p-1)(q-1)$. Select a random integer e, $1 < e < \phi$, such that $\gcd(e, \phi) = 1$.Use the extended Euclidean algorithm to compute the unique integer d, $1 < d < \phi$, such that $ed \equiv 1 \bmod \phi$. A's public key is (n, e); A's private key is d. Entity A signs message $m \in M$. Any entity B can verify A's signature and recover the message m from the signature.

Signature generation. A does following:
1: Compute $\tilde{m} = R(m)$, where $R(\cdots)$ may be a hash function
2: Compute $s = \tilde{m}^d \bmod n$
3: A's signature for m is s

Verification: B does following:
4: Obtain A's authentic public key (n,e)
5: Compute $\tilde{m} = s^e \bmod n$
6: Verify that $\tilde{m} = M_R \bmod n$, if not, reject the signature.
7: Recover that $m = R^{-1}(\tilde{m})$, where $R(\cdots)$ may be a hash function

Equation	*Computation complexity* $(x' = \mathrm{Log}_2\, x)$	*Min/Max*	*Min*	*Max*
$s = m'^d \bmod n$	$O(d'n'^2)$	$d \approx n$	$O(n'^3)$	$O(n'^3)$
$m' = s^e \bmod n$	$O(e'n'^2)$	$1 \leq e \leq n$	$O(n'^2)$	$O(n'^3)$

Table 25: Computation complexity in RSA signature and verification

A.3.2 The Digital Signature Algorithm (DSA)

The DSA signature scheme is presented in algorithm 21, and the complexity in table 26 and 27.In table 26 and 27 k', p' and q' represent the number of bit in k, p and q (which is equal to $(x' = Log_2\ x)$). Each entity creates an DSA public key and a corresponding private key, according to 21.

Algorithm 21 The Digital Signature Algorithm (DSA)

Key generation for DSA:
Each node creates a public key and corresponding private key. Each node A does the following. Select a prime number q such that $2^{159} < q < 2^{160}$. Choose t so that $0 < t \leq 8$, and select a prime number p $2^{511+64t} < p < 2^{512+64t}$ with the property that q divides $(p-1)$. Select an element $g \in Z_p^*$ and compute $\alpha = g^{(p-1)/q} \bmod p$ if $\alpha = 1$ go to the previous step and repeat until not equal. Select a random integer a such that $1 \leq a \leq q-1$. Compute $y = \alpha^a \bmod p$. A's public key is (p, q, α, y), A's private key is a.

DSA signature generation and verification:
Node A signs a binary message m of arbitrary length. Any node B can verify this signature by using A's public key.

Signature generation, node A does following:
1: Select a random integer k, $0 < k < q$
2: Compute $r = (\alpha^k \bmod p) \bmod q$
3: Compute $s = k^{-1}(h(m) + ar) \bmod q$
4: A's signature for m is the pair (r, s).

Verification: to verify A's signature (r, s) on m, B does following:
5: Obtain A's authentic public key (p, q, α, y)
6: Verify that $0 < r < q$ and $0 < s < q$, if not, then reject the signature.
7: Compute $w = s^{-1} \bmod q$ and h(m).
8: Compute $u_1 = wh(m) \bmod q$ and $u_2 = rw \bmod q$
9: Compute $v = (\alpha^{u_1} y^{u_2} \bmod p) \bmod q$
10: Accept the signature if and only if $v = r$.

Equation	*Computation complexity* $(x' = Log_2\ x)$	*Min/Max*	*Min*	*Max*
$r = (\alpha^k \bmod p) \bmod q$	$O(k'p'^2)$	$0 < k < q$	$O(p'^2)$	$O(q \cdot p'^2)$
$s = k^{-1} \bmod q$	$O(q'^3)$	-	$O(q'^3)$	$O(q'^3)$
$s = (k^{-1}h(m) + ar) \bmod q$	$O(2q'^2 + q')$	-	$O(q'^2)$	$O(q'^2)$
Initiator			$O(p'^2)$	$O(q \cdot p'^2)$

Table 26: Computation complexity of DSA signature

Before signing a message the signing part must select a random secret integer k, $0 < k < q$, and A's signature for m is the pair (r, s) as result of the algorithm.

Before doing the verification the receiver of the message has to obtain A's authentic public key (p, q, α, y) and verify that $0 < r < q$ and $0 < s < q$, if not, then reject the signature.

The verification accept the signature if and only if $v = r$.

Equation	*Computation complexity* ($x' = \text{Log}_2\, x$)	*Min/Max*	*Min*	*Max*
$w = s^{-1} \bmod q$ and h(m)	$O(q'^3)$	-	$O(q'^3)$	$O(q'^3)$
$u_1 = wh(m) \bmod q$	$O(q'^2)$	-	$O(q'^2)$	$O(q'^2)$
$u_2 = rw \bmod q$	$O(q'^2)$	-	$O(q'^2)$	$O(q'^2)$
$v = (\alpha^{u_1} y^{u_2} \bmod p) \bmod q$	$O(u_1' p'^2 + u_2' p'^2 + p'^2)$	-	$O(q \cdot p'^2)$	$O(q \cdot p'^2)$
Verification			$O(q \cdot p'^2)$	$O(q \cdot p'^2)$

Table 27: Computation complexity in DSA verification

A.3.3 The ElGamal Signature scheme

The ElGamal signature scheme is presented in algorithm 22, and the complexity in table 28 and 29. In table 28 and 29 k', $h(m)'$ and p' represent the number of bit in k, $h(m)$ and p (which is equal to $(x' = \text{Log}_2\, x)$). Each entity creates an ElGamal public key and a corresponding private key, according to 22.

Algorithm 22 The ElGamal Signature scheme

Key generation for ElGamal:
Each entity creates a public key and corresponding private key. Each entity A should do the following. Generate a large random prime number p and generator α of the multiplicative group Z_p^*. Select a random integer a, $1 \leq a \leq p-2$ and Compute $y = \alpha^a \bmod p$. A's public key is (p, α, y), A's private key is a. Before signature generation Select a random secret integer k, $1 \leq k \leq p-2$, with $\gcd(k, p-1) = 1$. Where A's signature for m is the pair (r, s).Before verification of the signature Verification B should Obtain A's authentic public key (p, α, y) and verify that $1 \leq r \leq p-1q$, if not, then reject the signature. B accept the signature if and only if $v_1 = v_2$.

ElGamal signature generation and verification:
Entity A signs a binary message m of arbitrary length. Any entity B can verify this signature by using A's public key.

Signature generation, entity A should do following:
1: Select a random secret integer k, $1 \leq k \leq p-2$, with $\gcd(k, p-1) = 1$.
2: Compute $r = (\alpha^k \bmod p)$
3: Compute $(k^{-1} \bmod (p-1))$
4: Compute $s = (k^{-1} h(m) - ar \bmod (p-1))$
5: A's signature for m is the pair (r, s).

Verification: to verify A's signature (r, s) on m, B should do the following:
6: Obtain A's authentic public key (p, α, y)
7: Verify that $1 \leq r \leq p-1q$, if not, then reject the signature.
8: Compute $v_1 = y^r r^s \bmod p$.
9: Compute h(m) and $v_2 = \alpha^{h(m)} \bmod p$.
10: Accept the signature if and only if $v_1 = v_2$.

Equation	*Computation complexity* $(x' = \text{Log}_2\, x)$	*Min/Max*	*Min*	*Max*
$r = (\alpha^k \bmod p)$	$O(k'p'^2)$	$1 \leq k \leq p-2$	$O(p'^2)$	$O(p'^3)$
$(k^{-1} \bmod (p-1))$	$O((p-1)'^3)$	-	$O((p-1)'^3)$	$O((p-1)'^3)$
$s = (k^{-1}h(m) - ar \bmod (p-1))$	$O(2k'(p-1)'^2 + (p-1)')$	-	$O((p-1)'^3)$	$O((p-1)'^3)$
Initiator			$O((p-1)'^3)$	$O(p'^3)$

Table 28: Computation complexity in ElGamal signature generation

Equation	*Computation complexity* $(x' = \text{Log}_2\, x)$	*Min/Max*	*Min*	*Max*
$v_1 = y^r r^s \bmod p$	$O(r'p'^2 + s'p'^2 + p'^2)$	$1 \leq r \leq p-1$	$O(p'^2)$	$O(p'^3)$
$h(m)$	-	-	-	-
$v_2 = \alpha^{h(m)} \bmod p$	$O(h(m)' \cdot p'^2)$	-	$O(h(m)' \cdot p'^2)$	$O(h(m)' \cdot p'^2)$
Verification			$O(h(m) \cdot' p'^2)$	$O(p'^3)$

Table 29: Computation complexity in ElGamal verification

A.3.4 The Schnorr Signature scheme

The Schnorr signature scheme is presented in algorithm 23, and the complexity in table 30 and 31. In table 30 and 31 k', p' and q' represent the number of bit in k, p and q (which is equal to $(x' = \text{Log}_2\, x)$). Each entity creates a public key and a corresponding private key, according to 23.

Equation	*Computation complexity* $(x' = \text{Log}_2\, x)$	*Min/Max*	*Min*	*Max*
$r = (\alpha^k \bmod p)$	$O(k'p'^2)$	$1 \leq k \leq q-1$	$O(p'^2)$	$O((q-1)' \cdot p'^2)$
$e = h(m\|r)$	–	-	-	-
$s = ae + k \bmod q$	$O(q'^2 + q')$	$h(m\|r)$	$O(q'^2)$	$O(q'^2)$
Initiator			$O(p'^2)$	$O((q-1)' \cdot p'^2)$

Table 30: Computation complexity in Schnorr signature generation

Algorithm 23 The Schnorr Signature scheme

Key generation:
Each entity creates a public key and corresponding private key. Each entity A should do the following. Select a prime number q (there is no constrain on the size of p and q). Select a prime number p with the property that q divides $(p-1)$. Select a generator α of the unique cyclic group of order q in Z_p^*. Select an element $g \in Z_p^*$ and compute $\alpha = g^{(p-1)/q} \bmod p$ if $\alpha = 1$ go to the previous step and repeat until not equal. Select a random integer a such that $1 \leq a \leq q-1$. Compute $y = \alpha^a \bmod p$. A's public key is (p, q, α, y), A's private key is a.

Signature generation, entity A should do following:
1: Select a random secret integer k, $1 \leq k \leq q-1$, with $\gcd(k, p-1) = 1$.
2: Compute $r = (\alpha^k \bmod p)$, $e = h(m||r)$ and $s = ae + k \bmod q$
3: A's signature for m is the pair (s, e).

Verification: to verify A's signature (r, s) on m, B should do the following:
4: Obtain A's authentic public key (p, q, α, y)
5: Compute $v = \alpha^s y^{-e} \bmod p$ and $e' = h(m||v)$.
6: Accept the signature if and only if $e' = e$.

Equation	*Computation complexity* $(x' = \mathrm{Log}_2\, x)$	*Min/Max*	*Min*	*Max*
$v = \alpha^s y^{-e} \bmod p$	$O(s'p'^2 + e'p'^2 +$	-	$O(q' \cdot p'^2)$	$O(q' \cdot p'^2)$
	$p'^2 + p'^3)$			
$e' = h(m\|v)$	-	-	-	
Verification			$O(q' \cdot p'^2)$	$O(q' \cdot p'^2)$

Table 31: Computation complexity in Schnorr signature verification

A.4 Key agreement based on asymmetric techniques

A.4.1 Station to station protocol (STS)

The Station to station protocol (STS) is a key agreement protocol with, mutual entity authentication, and explicit key authentication. The protocol is presented in algorithm 24, and the computation complexity in table 32. In table 32 x', y' and p' represent the number of bit in x, y and p (which is equal to $(x' = \mathrm{Log}_2\, x)$). Where E is a symmetric encryption algorithm, $S_A(m)$ is signed by A using the secret key. $S_A(m) = (H(M)^{d_A \bmod n_A})$ where $H(M) < n_A$. Select and public an appropriate prime p and generator α of Z_p^*, where $2 \leq \alpha \leq p-2$. Each user (entity) select RSA public and private keys, (e_A, n_A) and d_A with an off-line TTP the certificate can be included in eq. 2 and 3.

Algorithm 24 Station to station protocol (STS)

1: A generate a secret $1 \leq x \leq p-2$, B generate a secret $1 \leq y \leq p-2$ and compute $K = (\alpha^x)^y \bmod p$
2: $A \longrightarrow B:\ \ \alpha^x \bmod p$
3: $B \longrightarrow A:\ \ \alpha^y \bmod p, E_K(S_B(`\alpha^y, \alpha^x))$
4: $A \longrightarrow B:\ \ E_K(S_A(\alpha^x, \alpha^y))$

Equation	*Computation complexity* $(x' = \mathrm{Log}_2\, x)$	*Min/Max*	*Min*	*Max*
$\alpha^x \bmod p$	$O(x'p'^2)$	$1 \leq x \leq p-2$	$O(p'^2)$	$O((p-2)' \cdot p'^2)$
$\alpha^y \bmod p$	$O(y'p'^2)$	$1 \leq y \leq p-2$	$O(p'^2)$	$O((p-2)' \cdot p'^2)$
$K = (\alpha^x)^y \bmod p$	$O(x'p'^2 + y'p'^2 + p^2)$	$1 \leq x \leq p-2$ $1 \leq y \leq p-2$	$O(p'^2)$	$O((p-2)' \cdot p'^2)$
Initiator/verification			$O(p'^2)$	$O((p-2)' \cdot p'^2)$

Table 32: Computation complexity in Station to station protocol (STS)

A.4.2 X.509 Authentication protocol

This protocol has three different extensions; the long extensions is presented in algorithm 25.

Algorithm 25 X.509 Authentication protocol

1: Definition: $D_A = t_A, r_A, B, data_1^*, P_B(K_1^*)$, $D_B = t_B, r_B, A, r_B, data_2^*, P_A(K_2^*)$
2: $A \longrightarrow B:$ $Cert_A, D_A, Sign_A(D_A)$
3: $B \longrightarrow A:$ $Cert_B, D_B, Sign_B(D_B)$
4: $A \longrightarrow B:$ $r_B, B, Sign_A(r_B, B)$

A.4.3 Needham-Schroede public-key protocol

The Needham-Schroede public-key protocol is presented in algorithm 26, and Modified version in algorithm 27.

Algorithm 26 Needham-Schroede public-key protocol

1: $A \longrightarrow B:$ $P_B(K_1, A)$
2: $B \longrightarrow A:$ $P_A(K_1, K_2)$
3: $A \longrightarrow B:$ $P_B(K_2)$

Algorithm 27 Modified Needham-Schroede public-key protocol

1: $A \longrightarrow B:$ $P_B(K_1, A, r_1)$
2: $B \longrightarrow A:$ $P_A(K_1, r_1, r_2)$
3: $A \longrightarrow B:$ $P_B(r_2)$

A.5 Elliptic curves

According to [34]: Let p>3 be a prime. The elliptic curve $y^2 = x^3 + ax + b$ over Z_p is the set of solution $(x, y) \in Z_p \times Z_p$ to the congruence $y^2 \equiv x^3 + ax + b \pmod p$. Where $a, b \in Z_p$ are constant such that $4a^3 + 27b^2 \neq 0 \pmod p$, together with a special point O called the point at infinity. We have $P = (x_1, y_1)$ and $Q = (x_2, y_2)$, if $x_1 = x_2$ and $y_2 = -y_1$ then $P + Q = O$ otherwise $P + Q = (x_3, y_3)$

In addition: $x_3 = \lambda^2 - x_1 - x_2$ and $y_3 = \lambda(x_1 - x_3) - y_1$ where

$$\lambda = \begin{cases} (y_2 - y_1)(x_2 - x_1)^{-1} & \text{if } P \neq Q \\ (3x_1^2 + a)(2y_1)^{-1} & \text{if } P = Q \end{cases}$$

and $P + O = O + P = P$

Let lock at an example: Let the elliptic curve $y^2 = x^3 + x + 6$ over Z_P and that $\alpha = (2, 7)$, then $2\alpha = (2, 7) + (2, 7)$, in this case $P = Q$, and we have
$\lambda = ((3 \times 2^2 + 1)(2 \times 7)^{-1} \bmod 11) = 2 \times 3^{-1} \bmod 11 = 2 \times 4 \bmod 11 = 8$, then we have $x_3 = 8^2 - 2 - 2 \bmod 11 = 5$ and $y_3 = 8(2 - 5) - 7 \bmod 11 = 2$

That means that $2\alpha = (5, 2)$ for 3α we have that $3\alpha = (5, 2) + (2, 7)$, in this case $P \neq Q$ and we must use $P + Q = (x_3, y_3)$ and we have $\lambda = (7 - 2)(2 - 5)^{-1} \bmod 11 = 5 \times 8^{-1} \bmod 11 = 5 \times 7 \bmod 11 = 2$, where $x_3 = 2^2 - 5 - 2 \bmod 11 = 8$ and $y_3 = 2(5 - 8) - 2 \bmod 11 = 3$. Then $3\alpha = (8, 3)$

A.5.1 Elliptic Curve Digital Signature Algorithm (ECDSA)

The Elliptic Curve Digital Signature Algorithm (ECDSA) according to According to [34] is presented in algorithm 28.

Algorithm 28 Elliptic Curve Digital Signature Algorithm (ECDSA)

Initialisation:
Let p be a prime or power of two, and let E be an elliptic curve defined over F_p. Let A be a point on E having prime order q, such that the Discrete Logarithm problem in A is infeasible. Let $P = \{0,1\}^*$, $A = Zq^* x Zq^*$, and define: $K = \{(p, q, E, A, m, B) : B = mA\}$. Where $0 \leq m \leq q-1$. The values p, q, E, A and B are the public key, and m is the private key. For $K(p, q, E, A, m, B)$, and for a (secret) random number k, $1 \leq k \leq q-1$, define:

Signature generation:
1: $sign_K(x, k) = (r, s)$
2: $kA = (u, v)$
3: $r = u \bmod q$
4: $s = k^{-1}(SHA-1(x) + mr) \bmod q$

To be computed:
5: $x_3 = \lambda^2 - x_1 - x_2 \bmod p$
6: $y_3 = \lambda(x_1 - x_3) - y_1 \bmod p$
7: $\lambda = (y_2 - y_1)(x_2 - x_1) - 1 \bmod p$, if $P \neq Q$
8: $\lambda = (3x_1^2 + a)(2y_1) - 1 \bmod p$, if $P = Q$

Verification of signature:
9: $w = s^{-1} \bmod q$
10: $i = wSHA-1(x) \bmod q$
11: $j = wr \bmod q$
12: $(u, v) = iA + jB$

To be computed:
13: $x_3 = \lambda^2 - x_1 - x_2 \bmod p$
14: $y_3 = \lambda(x_1 - x_3) - y_1 \bmod p$
15: $\lambda = (y_2 - y_1)(x_2 - x_1) - 1 \bmod p$, if $P \neq Q$
16: $\lambda = (3x_1^2 + a)(2y_1) - 1 \bmod p$, if $P = Q$
17: $ver_K(x, (r, s)) = true$ if $u \bmod q = r$

A.6 Complexity

The computation complexity for some protocol is summarised and presented in table 33. Where the notation $n' = \log_2(n)$ and represent the number of bits in n, which is also used for p and q.

Protocol	*Min Sig*	*Max sign*	*Min Ver*	*Max Ver*
GQ id.	$O(n'^2)$	$O(n'^3)$	$O(n'^2)$	$O(n'^3)$
Schnorr id.	$O(p'^2)$	$O(q' \cdot p'^2)$	$O(p'^2)$	$O(q' \cdot p'^2)$
RSA sign.	$O(n'^3)$	$O(n'^3)$	$O(n'^2)$	$O(n'^3)$
DSA sign.	$O(p'^2)$	$O(q' \cdot p'^2)$	$O(q' \cdot p'^2)$	$O(q' \cdot p'^2)$
ElGamal sign.	$O((p-1)'^3)$	$O(p'^3)$	$O(h(m)' \cdot p'^3)$	$O(p'^3)$
Schnorr sign.	$O(p'^2)$	$O((q-1)' \cdot p'^2))$	$O(q' \cdot p'^2)$	$(q' \cdot p'^2)$
STS protocol	$O(p'^2)$	$O((p-2)' \cdot p'^2)$	$O(p'^2)$	$O((p-2)' \cdot p'^2)$

Table 33: Computation complexity in different protocols

B O-notation

According to [34] the complexity to calculate different argument is:

The general rule for the complexity to different calculation is summered in table, where $k = \log_2(\lfloor x \rfloor)$ and $l = \log_2(\lfloor y \rfloor)$ and assume that $k \geq l$:

ID	Expression	can be computed in
1	$x + y$	$O(k)$
2	$x - y$	$O(k)$
3	xy	$O(kl)$
4	$\lfloor x/y \rfloor$	$O(l(k-l))$ or $O(kl)$ is a weaker bound
5	$gcd(x, y)$	$O(k^3)$

Table 34: Computation complexity to different calculation

For operation in Z_n have the complexity to different calculation is summered in table, where k bits is $\log_2(n)$, and $0 \leq (m_1, m_2) \leq n - 1$:

ID	Expresion	O-notaion
1	$m_1 + m_2 \bmod n$	$O(k)$
2	$m_1 - m_2 \bmod n$	$O(k)$
3	$m_1 m_2 \bmod n$	$O(k^2)$
4	$m_1^{-1} \bmod n$	$O(k^3)$
5	$m_1^c \bmod n \; c < n$	$O((\log c)k^2)$

Table 35: Computation complexity to different calculation modular function

C Bit security level to different cipher type

The NIST Key Management Guideline [27] has some recommendation on key size for most preferred cryptographic algorithm. This is summary represented in table 36 :

Year	AES	SHA	DSA or Diffie-Hellman	RSA	ECDSA
Present	80	SHA-1 (160)	P=1024, S=160	512	113
Through 2010	80	SHA-224	P=1024, S=160	1024	160-223
Through 2030	112	SHA-224	P=2048, S=224	2048	224-255
Beyond 2030	128	SHA-256	P=3072, S=256	3072	256-383

Table 36: Recommended key size to different encryption algorithm and hash function

According to NIST, SHA-1 is demonstrated to provide less than 80 bits of security for digital signatures, so the use of SHA-1 is not recommended. To design new systems it is recommended to use one of the larger hash functions as SHA-224 or larger.

The next table 37 present comparable bit security between different cryptographic algorithms:

Bits of security	Symmetric key algorithms	DSA or Diffie-Hellman	RSA	ECDSA
80	2TDEA	P=1024, S=160	1024	160-223
112	3TDEA	P=2048, S=224	2048	224-255
128	AES-128	P=3072, S=256	3072	256-383
192	AES-192	P=7680, S=384	7680	384-511
256	AES-256	P=15360, S=512	15360	512+

Table 37: Equal security level depending on key size

D Program used to test the authentication protocol

```
/*
 *
 *This is the test program that is used to test the Authentication protocol
 *for MANET according to the Master Thesis. It is based on use of miracl
 *library and some example program from Miracl. I have used VC2005 that you may
 *get for free from microsoft, to use this program and installing the library
 *follow the instruction from Miracl and how to build a lib file on your computer.
 *I have used Windows XP version 5.1 service pack 2.
 *The program includes use of RSA, DSS (DSA), ECDSA (with a number of curves from
 *Miracl),SHA-1 and HMAC/SHA-1 (based on RFC 2202).As long as possible I have only
 *use instruction from ANSI C and Miracl.
 *
 *Ståle Jonny Berget sjberget@start.no (Master student at Gjøvik University College).
 *
*/

#include <stdio.h>
#include <time.h>
#include "miracl.h"
#include <sys/types.h>
#include <sys/timeb.h>

#define QBITS 160 /*to be used in DSA*/
#define PBITS 1024 /*to be used in DSA*/
#define chainl 20 /*the length of the hash chain*/
#define NP 2            /*To be used in RSA, use two primes - could be more */
#define PRIME_BITS 512  /*To be used in RSA, >=256 bits */
#define NRNODE 1 /*The number of nodes*/
#define NR 1000 /*the number of interaction for fast function*/
#define NR2 3 /*the number of interaction for slow function*/
#define NR3 10000 /*the number of interaction on messages*/
#define NR4 100 /*the number of interaction on messages that is slow*/

#define SHA_1_DIGESTSIZE 20
#define SHA_1_BLOCKSIZE 64
#define SHA_256_DIGESTSIZE  32
#define SHA_256_BLOCKSIZE   64
#define SHA_384_DIGESTSIZE  48
#define SHA_384_BLOCKSIZE   128
#define SHA_512_DIGESTSIZE  64
#define SHA_512_BLOCKSIZE   128
```

```
#define length_HASH_CHAIN 10
#define MAX_length_MASTER_HASH_CHAIN 10000
#define MAX_length_TRAFFIC_HASH_CHAIN 1000 /*must be smaller than
MAX_length_MASTER_HASH_CHAIN*/
#define MAX_NR_OF_SESSION 10
#define MAX_length_OF_SESSION_HASH_CHAIN 1000/*must be smaller than
MAX_length_MASTER_HASH_CHAIN*/

static miracl *mip;


/*Secp160 ECC Courve*/
char p_secp_160[]="FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF7FFFFFFF";
int a_secp_160=-3;
char b_secp_160[]="1C97BEFC54BD7A8B65ACF89F81D4D4ADC565FA45";
char q_secp_160[]="100000000000000000001F4C8F927AED3CA752257";
char x_secp_160[]="4A96B5688EF573284664698968C38BB913CBFC82";
char y_secp_160[]="23A628553168947D59DCC912042351377AC5FB32";

/*Secp 192 ECC Courve*/
char p_secp_192[]="FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFFFFFFFFFFFF";
int a_secp_192=-3;
char b_secp_192[]="64210519E59C80E70FA7E9AB72243049FEB8DEECC146B9B1";
char q_secp_192[]="FFFFFFFFFFFFFFFFFFFFFFFF99DEF836146BC9B1B4D22831";
char x_secp_192[]="188DA80EB03090F67CBF20EB43A18800F4FF0AFD82FF1012";
char y_secp_192[]="07192B95FFC8DA78631011ED6B24CDD573F977A11E794811";

/*Secp 224 ECC Courve*/
char p_secp_224[]="FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF000000000000000000000001";
int a_secp_224=-3;
char b_secp_224[]="B4050A850C04B3ABF54132565044B0B7D7BFD8BA270B39432355FFB4";
char q_secp_224[]="FFFFFFFFFFFFFFFFFFFFFFFFFFFF16A2E0B8F03E13DD29455C5C2A3D";
char x_secp_224[]="B70E0CBD6BB4BF7F321390B94A03C1D356C21122343280D6115C1D21";
char y_secp_224[]="BD376388B5F723FB4C22DFE6CD4375A05A07476444D5819985007E34";

/*EC Courve common from Miracle lib*/
char p_common_224[]="FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF000000000000000000000001";
int a_common_224=-3;
char b_common_224[]="B4050A850C04B3ABF54132565044B0B7D7BFD8BA270B39432355FFB4";
char q_common_224[]="FFFFFFFFFFFFFFFFFFFFFFFFFFFF16A2E0B8F03E13DD29455C5C2A3D";
char x_common_224[]="B70E0CBD6BB4BF7F321390B94A03C1D356C21122343280D6115C1D21";
char y_common_224[]="BD376388B5F723FB4C22DFE6CD4375A05A07476444D5819985007E34";

struct rsa_public_key {
big n; /*public key*/
big e; /*public key*/
```

```
};

struct rsa_private_key {
long seed; /*seed for RSA*/
long seed_RSA[2*NP]; /*seed for RSA*/
big p; /*private key*/
big q; /*private key*/
big d; /*private key*/
};

struct dsa_public_key {
big p; /*public key*/
big q; /*public key*/
big g; /*public key*/
big y; /*public key*/
big k; /*public key*/
};

struct dsa_private_key {
long seed; /*seed for DSA*/
big x; /*private key*/
};

struct ecdsa_curve {
big p;
    big a;
    big b;
    big q;
    big x;
    big y;
};

struct ecdsa_public_key {
big x; /*public key to signer*/
int ep; /*used in point comprestion*/
};

struct ecdsa_private_key {
long seed;
big d; /*private key of signer*/
};

struct rsa_sign{
big s;
};
struct ecdsa_and_dsa_sign {
```

```
big r; /*message vector*/
big s; /*message vector*/
};

struct signature{
struct rsa_sign sign_rsa;
struct ecdsa_and_dsa_sign sign_dsa;
struct ecdsa_and_dsa_sign sign_ecdsa;
};
struct challenge{
int challengeFromInitiatorInteger;
int challengeFromResponderInteger;
char challengeStringFromInitiator[SHA_1_DIGESTSIZE+1];
char challengeStringFromResponder[SHA_1_DIGESTSIZE+1];
char hashOfchallengeStringFromResponder[SHA_1_DIGESTSIZE+1];
char InitiatorsHashOfrespondsTochallengeFromResponder[SHA_1_DIGESTSIZE+1];
};

struct message{
big hash; /*hash value to the message*/
big sign; /*the signature to the hash*/
big verification; /*the verification of a signature*/
big encryption; /*encryption of a hash or smal message*/
big decryption; /*decryption of a encrypted hash or smal message*/
char message_array[512];
char mac[SHA_1_DIGESTSIZE+1];
struct challenge challenges;
char data_array[512];
};

struct certificate_Alg_Identifier {
int ALGORITHM_id;
char ALGORITHM_Type[20];
};

struct valid_time {
long long int notBefore;
long long int notAfter;
};

struct sub_pub_key_rsa {
char algorithm[20];
struct rsa_public_key public_key_rsa;
};

struct sub_pub_key_dsa {
```

```
char algorithm[20];
struct dsa_public_key public_key_dsa;
};

struct sub_pub_key_ecdsa {
char algorithm[20];
struct ecdsa_public_key public_key_ecdsa;
};

struct time_choice {
long long int utcTime;
long long int generalizedTime;
};
struct certificate_rsa {
big sign_certificate_rsa;
int version;
int CertificateSerialNumber;
struct certificate_Alg_Identifier AlgorithmIdentifier;
char SupportedAlgorithms[20];
struct valid_time Validity;
struct sub_pub_key_rsa SubjectPublicKeyInfo;
struct time_choice Time;
char issuer[20];
int subject;
big hash_certificate;
char AllCertificateElementThatIsSigned[1024];
};

struct certificate_dsa {
struct ecdsa_and_dsa_sign sign_certificate_dsa;
int version;
int CertificateSerialNumber;
struct certificate_Alg_Identifier AlgorithmIdentifier;
char SupportedAlgorithms[20];
struct valid_time Validity;
struct sub_pub_key_dsa SubjectPublicKeyInfo;
struct time_choice Time;
char issuer[20];
int subject;
big hash_certificate;
char AllCertificateElementThatIsSigned[1024];
};
struct certificate_ecdsa {
struct ecdsa_and_dsa_sign sign_certificate_ecdsa;
int version;
int CertificateSerialNumber;
```

```
struct certificate_Alg_Identifier AlgorithmIdentifier;
char SupportedAlgorithms[20];
struct valid_time Validity;
struct sub_pub_key_ecdsa SubjectPublicKeyInfo;
struct time_choice Time;
char issuer[20];
int subject;
big hash_certificate;
char AllCertificateElementThatIsSigned[1024];
};
struct certificate{
struct certificate_rsa rsa;
struct certificate_dsa dsa;
struct certificate_ecdsa ecdsa;
struct ecdsa_curve curve_ecdsa;
};

struct private_key{
struct rsa_private_key rsa;
struct dsa_private_key dsa;
struct ecdsa_private_key ecdsa;
};



struct node{
struct certificate ca_certificate;
struct certificate owen_certificate;
struct certificate others_certificate;
struct private_key owen_private_key;
long long int master_hash_key_time0;
long long int master_hash_key_end_life_time;
struct signature sign_master_hash_chain;
struct message transmit;
struct message receive;
char Master_hash_chain[MAX_length_MASTER_HASH_CHAIN][SHA_1_DIGESTSIZE+1];
char Traffic_hash_chain[MAX_length_TRAFFIC_HASH_CHAIN][SHA_1_DIGESTSIZE+1];
char session_hash_chain [MAX_NR_OF_SESSION]
[MAX_length_OF_SESSION_HASH_CHAIN]
[SHA_1_DIGESTSIZE+1];
char id_last_master_hash_time0_lifetime_in_array[100];
char hash_to_id_last_master_hash_time0_lifetime_array[SHA_1_DIGESTSIZE+1];
big hash_to_id_last_master_hash_time0_lifetime_big;
int val;
int index_kTraffic;
int index_session_key[MAX_NR_OF_SESSION];
```

```
};


long randise(void)
{ /* get a random number, input seed from keyboard */
    long seed;
    printf("Enter 9 digit random number seed  = ");
    scanf("%ld",&seed);
    getchar();
    return seed;
}

void strongp(big p,big pd,big ph,big pl,int n,long seed1,long seed2)
{ /* generate strong prime number =11 mod 12 suitable for RSA encryption */
    int r,r1,r2;

    irand(seed1);
    bigbits(2*n/3,pd);
    nxprime(pd,pd);
    expb2(n-1,ph);
    divide(ph,pd,ph);
    expb2(n-2,pl);
    divide(pl,pd,pl);
    subtract(ph,pl,ph);
    irand(seed2);
    bigrand(ph,ph);
    add(ph,pl,ph);
    r1=subdiv(pd,12,pl);
    r2=subdiv(ph,12,pl);
    r=0;
    while ((r1*(r2+r))%12!=5) r++;
    incr(ph,r,ph);
    do
    { /* find p=2*r*pd+1 = 11 mod 12 */
        multiply(ph,pd,p);
        premult(p,2,p);
        incr(p,1,p);
        incr(ph,12,ph);
    } while (!isprime(p));
}



void rsa_keygen(unsigned long *seed,big n, big p, big q, big e, big d)
{
```

```
/* RSA. Generate primes p & q. Use e=65537, and find d=1/e mod (p-1)(q-1)*/
/*this is key generation for rsa it's generate "p","q" and compute "n"/"phi",
/*and input "e" to compute "d" where n=pq, phi=(p-1)(q-1) and d=1/e mod phi or
/*ed=1 mod phi public key is (n,e) and private key is (p,q,d)*/

char stack_mem[mr_big_reserve(4,50)];
big phi,p1,q1,t;
    memset(stack_mem,0,mr_big_reserve(4,50));

    phi=mirvar_mem(stack_mem,0);
    p1=mirvar_mem(stack_mem,1);
    q1=mirvar_mem(stack_mem,2);
    t=mirvar_mem(stack_mem,3);

irand(*seed);
   do
    {
        bigbits(PRIME_BITS,p);
        if (subdivisible(p,2)) incr(p,1,p);
        while (!isprime(p)) incr(p,2,p);

        bigbits(PRIME_BITS,q);
        if (subdivisible(q,2)) incr(q,1,q);
        while (!isprime(q)) incr(q,2,q);

        multiply(p,q,n);      /* n=p.q */

        lgconv(65537L,e);
        decr(p,1,p1);
        decr(q,1,q1);
        multiply(p1,q1,phi);  /* phi =(p-1)*(q-1) */
    } while (xgcd(e,phi,d,d,t)!=1);
}

void rsa_encrypt(big n, big p, big q, big e, big d, big hash, big c)
{
/* this is the rsa encryption for bigvariable*/
char stack_mem[mr_big_reserve(4,50)];
    big dp,dq,p1,q1;
    memset(stack_mem,0,mr_big_reserve(4,50));

dp=mirvar_mem(stack_mem,0);
dq=mirvar_mem(stack_mem,1);
    p1=mirvar_mem(stack_mem,2);
    q1=mirvar_mem(stack_mem,3);
```

```
/*initialisation of parameters*/
copy(d,dp); /*dp=d*/
    copy(d,dq); /*dq=dp*/
decr(p,1,p1); /*p1=p-1*/
    decr(q,1,q1); /*q1=q-1*/

/*-------------------------*/
divide(dp,p1,p1);   /* dp=d mod p-1 */
divide(dq,q1,q1);   /* dq=d mod q-1 */

/*encryption of the plain text "hash" to chipher text "c"*/
powmod(hash,e,n,c);
}

void rsa_decrypt(big n, big p, big q, big e, big d, big m , big c)
{
/* this is the rsa decryption bigvariable. it use chinese remainder thereom */
big_chinese ch;
char stack_mem[mr_big_reserve(7,50)];
    big p1,q1,c1,dp,dq,pm[2];
    memset(stack_mem,0,mr_big_reserve(7,50));

    dp=mirvar_mem(stack_mem,0);
dq=mirvar_mem(stack_mem,1);
    pm[0]=mirvar_mem(stack_mem,2);
pm[1]=mirvar_mem(stack_mem,3);
c1=mirvar_mem(stack_mem,4);
p1=mirvar_mem(stack_mem,5);
q1=mirvar_mem(stack_mem,6);

/*initialisation of chinese remainder thereom*/
pm[0]=p;
    pm[1]=q;
crt_init(&ch,2,pm);

/*initialisation of parameters*/
copy(c,c1); /*c-->c1*/
copy(d,dp); /*d-->dp*/
    copy(d,dq); /*d-->dq*/
decr(p,1,p1); /*p1=p-1*/
    decr(q,1,q1); /*q1=q-1*/

/*-------------------------*/
divide(dp,p1,p1);   /* dp=d mod p-1 */
divide(dq,q1,q1);   /* dq=d mod q-1 */
```

```
/*-------------------------*/
powmod(c1,dp,p,pm[0]);    /* get result mod p */
    powmod(c1,dq,q,pm[1]);    /* get result mod q */

/*retrive the plain text*/
crt(&ch,pm,m);           /* combine them using CRT */

crt_end(&ch); /*clean-up after chinese remainder thereom*/
}

void rsa_signature(big *n, big *p, big *q, big *e, big *d, big *sign, big *hash)
{
/* this is the rsa signature it use chinese remainder thereom */
big_chinese ch;
char stack_mem[mr_big_reserve(9,50)];
    big p_1,q_1,p1,q1,hash1,dp,dq,pm[2];
    memset(stack_mem,0,mr_big_reserve(9,50));

    dp=mirvar_mem(stack_mem,0);
dq=mirvar_mem(stack_mem,1);
    pm[0]=mirvar_mem(stack_mem,2);
pm[1]=mirvar_mem(stack_mem,3);
hash1=mirvar_mem(stack_mem,4);
p1=mirvar_mem(stack_mem,5);
q1=mirvar_mem(stack_mem,6);
p_1=mirvar_mem(stack_mem,7);
q_1=mirvar_mem(stack_mem,8);

/*initialisation of chinese remainder thereom*/
copy(*p,p_1); /*p-->p_1*/
copy(*q,q_1); /*q-->q_1*/
pm[0]=p_1;
    pm[1]=q_1;
crt_init(&ch,2,pm);

/*initialisation of parameters*/
copy(*hash,hash1); /*hash-->hash1*/
copy(*d,dp); /*d-->dp*/
    copy(*d,dq); /*d-->dq*/
decr(p_1,1,p1); /*p1=p_1-1*/
    decr(q_1,1,q1); /*q1=q_1-1*/

/*-------------------------*/
divide(dp,p1,p1);   /* dp=d mod p-1 */
divide(dq,q1,q1);   /* dq=d mod q-1 */
```

```
/*-------------------------*/
powmod(hash1,dp,p_1,pm[0]);    /* PM[0]=hash1^dp mod p */
    powmod(hash1,dq,q_1,pm[1]);    /* PM[1]=hash1^dq mod q */

/*sign the hash value*/
crt(&ch,pm,*sign);           /* combine them using CRT */

crt_end(&ch); /*clean-up after chinese remainder thereom*/
}

void rsa_sign_verification(big *n, big *e, big *sign, big *hash, int *val)
{
/* this is the rsa verification on rsa signature that use bigvariable*/
int test;

char stack_mem[mr_big_reserve(1,50)];
    big hash1;
    memset(stack_mem,0,mr_big_reserve(1,50));

/*initialisation on big variable and test paramter*/
hash1=mirvar_mem(stack_mem,0);
test=0;

/*The verification on signature (sign) from the call to this function*/
powmod(*sign,*e,*n,hash1); /* hash1=sign^e nod n*/
if (test!=compare(hash1,*hash))
*val=0;
else
*val=1;
}
/*void ecdsa_read_courve(big a, big b, big p, big q, big x, big y)
{
    FILE *fp;
    int ep,bits;
    miracl *mip;

    fp=fopen("common224.ecs","rt");
    if (fp==NULL)
    {
        printf("file common.ecs does not exist\n");
        return 0;
    }
    fscanf(fp,"%d\n",&bits);
```

```
    mip=mirsys(bits/4,16); */ /* Use Hex internally */
/*a=mirvar(0);
    b=mirvar(0);
    p=mirvar(0);
    q=mirvar(0);
    x=mirvar(0);
    y=mirvar(0);

    innum(p,fp);
    innum(a,fp);
    innum(b,fp);
    innum(q,fp);
    innum(x,fp);
    innum(y,fp);
    fclose(fp);
}*/

void ecdsa_gen(big *a,big *b,big *p,big *q,big *x,big *y,
   big *d,long *seed,big *x_new,int *ep)
{
epoint *g,*w;


/*initialisation, to keep x-value from ecc courve for later use*/
/*this function is returning both the old one x and new x*/
copy(*x,*x_new); /*x_new=x as x-->x_new*/

/* randomise */
    irand(*seed);

    ecurve_init(*a,*b,*p,MR_PROJECTIVE);  /* initialise curve */

    g=epoint_init();
    w=epoint_init();

    if (!epoint_set(*x_new,*y,0,g)) /* initialise point of order q */
    {
        printf("Problem - point (x,y) is not on the curve\n");
    }

    ecurve_mult(*q,g,w);
    if (!point_at_infinity(w))
    {
        printf("Problem - point (x,y) is not of order q\n");
    }
```

```
/* generate new public/private keys based on the ecc courv */
/* x_new-public key and d-private key*/

/*generation of private key (d)*/
    bigrand(*q,*d);

    /*compute public key (x_new) based on private key d*/
ecurve_mult(*d,g,g);
    *ep=epoint_get(g,*x_new,*x_new); /* compress point */
}

void ecdsa_sign(big *a,big *b,big *p,big *q,big *x,big *y,big *d,long *seed,
big *x_new,big *r,big *s,big *hash)
{
    epoint *g;

char stack_mem[mr_big_reserve(1,50)];
    big k;
    memset(stack_mem,0,mr_big_reserve(1,50));
k=mirvar_mem(stack_mem,0);

/* randomise */
    irand(*seed);

    ecurve_init(*a,*b,*p,MR_PROJECTIVE);  /* initialise curve */
    g=epoint_init();
    epoint_set(*x,*y,0,g); /* initialise point of order q */

/* calculate r - this can be done offline,
   and hence amortized to almost nothing   */
    bigrand(*q,k);
    ecurve_mult(k,g,g);      /* see ebrick.c for method to speed this up */
    epoint_get(g,*r,*r);
    divide(*r,*q,*q);

/* calculate s */
    xgcd(k,*q,k,k,k);

    mad(*d,*r,*hash,*q,*q,*s);
    mad(*s,k,k,*q,*q,*s);
}

void ecdsa_ver(big *a,big *b,big *p,big *q,big *x,big *y,big *d,long *seed,
   big *x_new,int *ep,big *r,big *s,big *hash, int *val)
{
    epoint *g,*public;
```

```
    char stack_mem[mr_big_reserve(6,50)];
    big ss,v,u1,u2,xx,yy;
    memset(stack_mem,0,mr_big_reserve(6,50));
v=mirvar_mem(stack_mem,0);
    u1=mirvar_mem(stack_mem,1);
    u2=mirvar_mem(stack_mem,2);
xx=mirvar_mem(stack_mem,3);
yy=mirvar_mem(stack_mem,4);
    ss=mirvar_mem(stack_mem,5);

/*copy of s from the call so it doesn't chang during the function*/
copy(*s,ss);

/*initialisation with curve parameters use x from the curve */
/*(don't use x_new that's signers public key*/
    ecurve_init(*a,*b,*p,MR_PROJECTIVE);  /* initialise curve */
    g=epoint_init();
    epoint_set(*x,*y,0,g); /* initialise point of order q */

/*use signer public key x_new*/
    public=epoint_init();

    epoint_set(*x_new,*x_new,*ep,public);  /* decompress */

    if (compare(*r,*q)>=0 || compare(ss,*q)>=0)
*val=0;
    xgcd(ss,*q,ss,ss,ss);
    mad(*hash,ss,ss,*q,*q,u1);
    mad(*r,ss,ss,*q,*q,u2);

    ecurve_mult2(u2,public,u1,g,g);
    epoint_get(g,v,v);
    divide(v,*q,*q);
    if (compare(v,*r)==0)
*val=1;
else
*val=0;
}

void strip(char *name)
/* strip off filename extension */
{
    int i;
    for (i=0;name[i]!='\0';i++)
    {
```

```
        if (name[i]!='.') continue;
        name[i]='\0';
        break;
    }
}

static void hashing_sha_1(char *hash_inputPtr,int *l,
  char hash_output[SHA_1_DIGESTSIZE+1])
/* compute hash function based on SHA-1,hash_input is the input,*/
/*l-the lengt of input,hash_output-is the hash value output*/
{
    sha sh;

    shs_init(&sh);
    for ( ; *hash_inputPtr!='\0'; hash_inputPtr++)
shs_process(&sh,*hash_inputPtr);
    shs_hash(&sh,hash_output);
hash_output[SHA_1_DIGESTSIZE]='\0';
}

void hash_chain_sha_1(char h0[SHA_1_DIGESTSIZE+1], int *cl,
  char hashchain[length_HASH_CHAIN][SHA_1_DIGESTSIZE+1])
/*build hash chain based on SHA-1, h0 is the initial value, cl-the length of */
/*the chain,hashchain- is the chain as an 2 dim array  */
{
int l,j;

/*Initialisation*/
l=SHA_1_DIGESTSIZE;

strcpy(hashchain[0],h0);

/*generering av hash chain*/
for (j=0;j<(*cl-1);j++)
hashing_sha_1(hashchain[j],&l, hashchain[j+1]);
}

void pr_sha(FILE* fp, char* s, int t)
/* Function to print the digest */
{
int     i ;
fprintf(fp, "0x");
for (i = 0 ; i < t ; i++)
fprintf(fp, "%02x", s[i]);
fprintf(fp, "0x");
}
```

```
void truncate(char* d1, char* d2, int len)
/* char*   d1-data to be truncated
   char*   d2-truncated data
   int     len-length in bytes to keep
*/
{
int     i ;

for (i = 0 ; i < len ; i++)
d2[i] = d1[i];
}


void hmac_sha_1(char *k, int *lk, char *d, int *ld, char *out, int *t)
   /*k-secret key, lk-length of the key in bytes, d-data, ld-length of data in*/
   /*bytes, out-output buffer, at least "t" bytes, int t */
   /*Function to compute the digest */
   /*Based on RFC 2202 Test Cases for HMAC-MD5 and HMAC-SHA-1 September 1997*/
 {
char isha[SHA_1_DIGESTSIZE], osha[SHA_1_DIGESTSIZE];
char key[SHA_1_DIGESTSIZE];
char buf[SHA_1_BLOCKSIZE];
char k_temp[SHA_1_DIGESTSIZE];
int i,l_key;

sha sh; /*to be used for SHA-1*/
/*sha256 sh32; to be used for SHA-256*/
/*sha512 sh64; to be used for SHA-512*/

/*Copy of key "k" and length of key "lk" to be used in this function to keep
this value*/
l_key=*lk;
for (i=0; i<l_key; i++)
k_temp[i]=k[i];

if (l_key > SHA_1_BLOCKSIZE)
{
shs_init(&sh);
for (i=0;i<l_key;i++)
shs_process(&sh,k_temp[i]);
shs_hash(&sh,key);
/*copy from key -->k_temp*/
for (i=0; i<l_key; i++) /* k = key*/
k_temp[i]=key[i];
l_key = SHA_1_DIGESTSIZE;
```

```
 }

 /**** Inner Digest ****/
 shs_init(&sh);

 /* Pad the key for inner digest */
for (i = 0 ; i < l_key ; ++i)
 buf[i] = k_temp[i] ^ 0x36;
for (i = l_key ; i < SHA_1_BLOCKSIZE ; ++i)
 buf[i] = 0x36;


 /*input data to sha function*/
 for (i=0;i<SHA_1_BLOCKSIZE;i++)
 shs_process(&sh,buf[i]);
 for (i=0;i<*ld;i++)
 shs_process(&sh,d[i]);
 shs_hash(&sh,isha);

 /**** Outer Digest ****/
 shs_init(&sh);

 /* Pad the key for outer digest */
 for (i = 0 ; i < l_key ; ++i)
 buf[i] = k_temp[i] ^ 0x5C;
 for (i = l_key; i < SHA_1_BLOCKSIZE; ++i)
 buf[i] = 0x5C;

 /*input data to sha function*/
 for (i=0;i<SHA_1_BLOCKSIZE;i++)
 shs_process(&sh,buf[i]);
 for (i=0;i<SHA_1_DIGESTSIZE;i++)
 shs_process(&sh,isha[i]);
 shs_hash(&sh,osha);

 /* truncate and print the results */
 *t = *t > SHA_1_DIGESTSIZE ? SHA_1_DIGESTSIZE : *t;
 truncate(osha, out, *t);
 out[*t]='\0';

}


void dss_setup(big p, big q, big g, long seeddssetup)
/*Use pointer to the value*/
{
```

```
char stack_mem[mr_big_reserve(4,50)];
    big h,n,s,t;
    memset(stack_mem,0,mr_big_reserve(4,50));

    h=mirvar_mem(stack_mem,0);
    n=mirvar_mem(stack_mem,1);
    s=mirvar_mem(stack_mem,2);
    t=mirvar_mem(stack_mem,3);

/* randomise */
irand(seeddssetup);

/* generate q */
    forever
    {
        bigbits(QBITS,q);
        nxprime(q,q);
        if (logb2(q)>QBITS) continue;
        break;
    }

/* generate p */
    expb2(PBITS,t);
    decr(t,1,t);
    premult(q,2,n);
    divide(t,n,t);
    expb2(PBITS-1,s);
    decr(s,1,s);
    divide(s,n,s);
    forever
    {
        bigrand(t,p);
        if (compare(p,s)<=0) continue;
        premult(p,2,p);
        multiply(p,q,p);
        incr(p,1,p);
        copy(p,n);
        if (isprime(p)) break;
    }

/* generate g */
    do {
        decr(p,1,t);
        bigrand(t,h);
        divide(t,q,t);
        powmod(h,t,p,g);
```

```
    } while (size(g)==1);
}
void dss_gen(big p, big q, big g, big x, big y, long seeddssgen)
/*Use pointer to the value*/
{
    /* randomise */
irand(seeddssgen);

/*generation public key*/
powmod(g,q,p,y);

if (size(y)!=1)
        printf("Problem - generator g is not of order q\n");

/* generate public/private keys */
bigrand(q,x);
    powmod(g,x,p,y);
}

void dss_sign(big p,big q,big g,big x,big hash,big r,big s,long seed)
{
/*initialisation of new big variables*/
char stack_mem[mr_big_reserve(1,50)];
    big k;
    memset(stack_mem,0,mr_big_reserve(1,50));
    k=mirvar_mem(stack_mem,0);

/* randomise */
    irand(seed);
/* calculate r - this can be done offline,  and hence amortized to almost
nothing  */
bigrand(q,k);
    powmod(g,k,p,r);  /* see brick.c for method to speed this up */
divide(r,q,q);

/* calculate s */
xgcd(k,q,k,k,k);
    mad(x,r,hash,q,q,s);
    mad(s,k,k,q,q,s);
}

void dss_ver(big p,big q,big g,big y,big r,big s,big hash, int *val)
{
    /*initialisation of new big variables*/
char stack_mem[mr_big_reserve(4,50)];
    big v,u1,u2,sinv;
```

```
    memset(stack_mem,0,mr_big_reserve(4,50));
    v=mirvar_mem(stack_mem,0);
u1=mirvar_mem(stack_mem,1);
u2=mirvar_mem(stack_mem,2);
sinv=mirvar_mem(stack_mem,3);

/*copy s to sinv to prepear for computation of sinv by xgcd later,
if not the return value of s from function will changes to inv of s*/
copy(s,sinv);


/*initial verification of signature*/
    if (compare(r,q)>=0 || compare(s,q)>=0)
*val=0;
else
{
xgcd(sinv,q,sinv,sinv,sinv); /*compute invers to s, (sinv=1/s)*/
mad(hash,sinv,sinv,q,q,u1);
mad(r,sinv,sinv,q,q,u2);
powmod2(g,u1,y,u2,p,v);
divide(v,q,q);
if (compare(v,r)==0)
*val=1;
else
*val=0;
}
}


void hash_laste_master_hash_chain_key
(int nodeID,
 char master_hash_chain_last_key[SHA_1_DIGESTSIZE+1],
 long long int start_time,long long int end_time,
 char result[100],big hash)
{
char buffer1[20], buffer2[20],buffer3[20],temp_hash1[SHA_1_DIGESTSIZE+1];
int l;

l=0;

_i64toa(start_time,buffer1,16);
_i64toa(end_time,buffer2,16);
_itoa(nodeID,buffer3,16);
strcpy(result,buffer3);
strcat(result,master_hash_chain_last_key);
```

```
strcat(result,buffer1);
strcat(result,buffer2);
hashing_sha_1(result,&l,temp_hash1);
bytes_to_big(SHA_1_DIGESTSIZE,temp_hash1,hash);
}

void hash_of_rsa_certificate( int version,/* Buffer nr 1*/
int CertificateSerialNumber,/* Buffer nr 2*/
int ALGORITHM_id,/* Buffer nr 3*/
char ALGORITHM_Type[],/* Buffer nr 4*/
char issuer[],/* Buffer nr 5*/
long long int notBefore,/* Buffer nr 6*/
long long int notAfter,/* Buffer nr 7*/
int subject,/* Buffer nr 8*/
char algorithm[],/* Buffer nr 9*/
big n,/* Buffer nr 10*/
big e,/* Buffer nr 11*/
big hash_certificate,
char result[])
{
char buffer1[20],buffer2[20],buffer3[20],buffer4[20],buffer5[20];
char buffer6[20],buffer7[20],buffer8[20],buffer9[20];
char buffer10[200],buffer11[200];
char temp_hash1[SHA_1_DIGESTSIZE+1];
int l,len1,len2;
int max;

len1=0;
len2=0;
l=0;
max=200;

_itoa(version,buffer1,16);
_itoa(CertificateSerialNumber,buffer2,16);
_itoa(ALGORITHM_id,buffer3,16);
_i64toa(notBefore,buffer6,16);
_i64toa(notAfter,buffer7,16);
_itoa(subject,buffer8,16);
strcpy(buffer4,ALGORITHM_Type);
strcpy(buffer5,issuer);
strcpy(buffer9,algorithm);
len1=big_to_bytes(max,n,buffer10,FALSE);
buffer10[len1]='\0';
len2=big_to_bytes(max,e,buffer11,FALSE);
buffer11[len2]='\0';
```

```
strcpy(result,buffer1);
strcat(result,buffer2);
strcat(result,buffer3);
strcat(result,buffer4);
strcat(result,buffer5);
strcat(result,buffer6);
strcat(result,buffer7);
strcat(result,buffer8);
strcat(result,buffer9);
strcat(result,buffer10);
strcat(result,buffer11);
hashing_sha_1(result,&l,temp_hash1);
bytes_to_big(SHA_1_DIGESTSIZE,temp_hash1,hash_certificate);
}


void hash_of_dsa_certificate( int version,/* Buffer nr 1*/
int CertificateSerialNumber,/* Buffer nr 2*/
int ALGORITHM_id,/* Buffer nr 3*/
char ALGORITHM_Type[],/* Buffer nr 4*/
char issuer[],/* Buffer nr 5*/
long long int notBefore,/* Buffer nr 6*/
long long int notAfter,/* Buffer nr 7*/
int subject,/* Buffer nr 8*/
char algorithm[],/* Buffer nr 9*/
big g,/* Buffer nr 10*/
big k,/* Buffer nr 11*/
big p,/* Buffer nr 12*/
big q,/* Buffer nr 13*/
big y,/* Buffer nr 14*/
big hash_certificate,
char result[])
{
char buffer1[20],buffer2[20],buffer3[20],buffer4[20],buffer5[20];
char buffer6[20],buffer7[20],buffer8[20],buffer9[20];
char buffer10[200],buffer11[200],buffer12[200],buffer13[200],buffer14[200];
char temp_hash1[SHA_1_DIGESTSIZE+1];
int l,len;
int max;

l=0;
max=200;

_itoa(version,buffer1,16);
_itoa(CertificateSerialNumber,buffer2,16);
_itoa(ALGORITHM_id,buffer3,16);
```

```
_i64toa(notBefore,buffer6,16);
_i64toa(notAfter,buffer7,16);
_itoa(subject,buffer8,16);
strcpy(buffer4,ALGORITHM_Type);
strcpy(buffer5,issuer);
strcpy(buffer9,algorithm);
len=big_to_bytes(max,g,buffer10,FALSE);
buffer10[len]='\0';
len=big_to_bytes(max,k,buffer11,FALSE);
buffer11[len]='\0';
len=big_to_bytes(max,p,buffer12,FALSE);
buffer12[len]='\0';
len=big_to_bytes(max,q,buffer13,FALSE);
buffer13[len]='\0';
len=big_to_bytes(max,y,buffer14,FALSE);
buffer14[len]='\0';
strcpy(result,buffer1);
strcat(result,buffer2);
strcat(result,buffer3);
strcat(result,buffer4);
strcat(result,buffer5);
strcat(result,buffer6);
strcat(result,buffer7);
strcat(result,buffer8);
strcat(result,buffer9);
strcat(result,buffer10);
strcat(result,buffer11);
strcat(result,buffer12);
strcat(result,buffer13);
strcat(result,buffer14);
hashing_sha_1(result,&l,temp_hash1);
bytes_to_big(SHA_1_DIGESTSIZE,temp_hash1,hash_certificate);
}


void hash_of_ecdsa_certificate( int version,/* Buffer nr 1*/
int CertificateSerialNumber,/* Buffer nr 2*/
int ALGORITHM_id,/* Buffer nr 3*/
char ALGORITHM_Type[],/* Buffer nr 4*/
char issuer[],/* Buffer nr 5*/
long long int notBefore,/* Buffer nr 6*/
long long int notAfter,/* Buffer nr 7*/
int subject,/* Buffer nr 8*/
char algorithm[],/* Buffer nr 9*/
big x,/* Buffer nr 10*/
int ep,/* Buffer nr 11*/
```

```
big hash_certificate,
char result[])
{
char buffer1[20],buffer2[20],buffer3[20],buffer4[20],buffer5[20];
char buffer6[20],buffer7[20],buffer8[20],buffer9[20],buffer11[20];
char buffer10[100];
char temp_hash1[SHA_1_DIGESTSIZE+1];
int l,len;
int max;

l=0;
max=100;

_itoa(version,buffer1,16);
_itoa(CertificateSerialNumber,buffer2,16);
_itoa(ALGORITHM_id,buffer3,16);
_i64toa(notBefore,buffer6,16);
_i64toa(notAfter,buffer7,16);
_itoa(subject,buffer8,16);
_itoa(ep,buffer11,16);
strcpy(buffer4,ALGORITHM_Type);
strcpy(buffer5,issuer);
strcpy(buffer9,algorithm);
len=big_to_bytes(max,x,buffer10,FALSE);
buffer10[len]='\0';
strcpy(result,buffer1);
strcat(result,buffer2);
strcat(result,buffer3);
strcat(result,buffer4);
strcat(result,buffer5);
strcat(result,buffer6);
strcat(result,buffer7);
strcat(result,buffer8);
strcat(result,buffer9);
strcat(result,buffer10);
strcat(result,buffer11);
hashing_sha_1(result,&l,temp_hash1);
bytes_to_big(SHA_1_DIGESTSIZE,temp_hash1,hash_certificate);
}


void message1_rsa( char owenAllCertificateElementThatIsSigned[],
big signatureToTheCertificate,
char masterHashChainIdTimeStartStop[],
big sigMasterHashChainIdTimeStartStop,
char challengeString[],
```

```
int challengeInteger,
char kMasterLastUsed[],
char kTrafficLastUsed[],
char kMasterNext[],
char macToChallengekMasterkTraffic[],
char message_1[])
{
int len1,len2,max;
char buffer1[200],buffer2[200],buffer5[200],buffer6[SHA_1_DIGESTSIZE+1];
int lk_hmac_sha_1; /*The key size to k_hmac_sha_1 in byte*/
int ld_hmac_sha_1; /*the length of data input in byte*/
int t_hmac_sha_1; /*The size of output from HMAC in byte*/

lk_hmac_sha_1=20;
t_hmac_sha_1=20;
max=200;

strcpy(challengeString,"01234567890123456789");
len1=big_to_bytes(max,signatureToTheCertificate,buffer1,FALSE);
buffer1[len1]='\0';
len2=big_to_bytes(max,sigMasterHashChainIdTimeStartStop,buffer2,FALSE);
buffer2[len2]='\0';
strcpy(message_1,owenAllCertificateElementThatIsSigned);
strcat(message_1,buffer1);
strcat(message_1,masterHashChainIdTimeStartStop);
strcat(message_1,buffer2);
strcat(message_1,challengeString);
strcat(message_1,kMasterLastUsed);
strcat(message_1,kTrafficLastUsed);
strcpy(buffer5,challengeString);
strcat(buffer5,kTrafficLastUsed);
for (ld_hmac_sha_1=0; buffer5[ld_hmac_sha_1]!='\0';ld_hmac_sha_1++);
hmac_sha_1(kMasterNext,&lk_hmac_sha_1,buffer5,&ld_hmac_sha_1,buffer6,&t_hmac_sha_1);
strcat(message_1,buffer6);
}

void message2_rsa(char owenAllCertificateElementThatIsSigned[],
big signatureToTheCertificate,
char masterHashChainIdTimeStartStop[],
big sigMasterHashChainIdTimeStartStop,
char challengeStringToInitiator[],
int challengeToInitiatorInt,
char kMasterLastUsed[],
char kTrafficLastUsed[],
char kMasterNext[],
char macToChallengekMasterkTraffic[],
```

```
char message_2[],
char challengeFromInitiator[],
char hashOfchallengeToInitiator[])
{
int len1,len2,max,l;
char buffer1[200],buffer2[200],buffer5[200],buffer6[SHA_1_DIGESTSIZE+1];
int lk_hmac_sha_1; /*The key size to k_hmac_sha_1 in byte*/
int ld_hmac_sha_1; /*the length of data input in byte*/
int t_hmac_sha_1; /*The size of output from HMAC in byte*/

lk_hmac_sha_1=20;
t_hmac_sha_1=20;
max=200;
l=0;

len1=big_to_bytes(max,signatureToTheCertificate,buffer1,FALSE);
buffer1[len1]='\0';
len2=big_to_bytes(max,sigMasterHashChainIdTimeStartStop,buffer2,FALSE);
buffer2[len2]='\0';
strcpy(message_2,owenAllCertificateElementThatIsSigned);
strcat(message_2,buffer1);
strcat(message_2,masterHashChainIdTimeStartStop);
strcat(message_2,buffer2);


/*Establish the challenge to initiator and hash it*/
strcpy(challengeStringToInitiator,"9");
strcat(challengeStringToInitiator,challengeFromInitiator);
hashing_sha_1(challengeStringToInitiator,&l,hashOfchallengeToInitiator);

/*include the challenge in the message*/
strcat(message_2,hashOfchallengeToInitiator);
strcat(message_2,kMasterLastUsed);
strcat(message_2,kTrafficLastUsed);

/*generate MAC*/
strcpy(buffer5,hashOfchallengeToInitiator);
strcat(buffer5,kTrafficLastUsed);
for (ld_hmac_sha_1=0; buffer5[ld_hmac_sha_1]!='\0';ld_hmac_sha_1++);
hmac_sha_1(kMasterNext,&lk_hmac_sha_1,buffer5,&ld_hmac_sha_1,
buffer6,&t_hmac_sha_1);

strcat(message_2,buffer6);
}
```

```
void message1_ecdsa_dsa(char owenAllCertificateElementThatIsSigned[],
big signatureToTheCertificate_r,
big signatureToTheCertificate_s,
char masterHashChainIdTimeStartStop[],
big sigMasterHashChainIdTimeStartStop_r,
big sigMasterHashChainIdTimeStartStop_s,
char challengeString[],
int challengeInteger,
char kMasterLastUsed[],
char kTrafficLastUsed[],
char kMasterNext[],
char macToChallengekMasterkTraffic[],
char message_1[])
{
int len1,len2,len3,len4,max;
char buffer1[200],buffer2[200],buffer3[200],buffer4[200],buffer5[200];
char buffer6[SHA_1_DIGESTSIZE+1];
int lk_hmac_sha_1; /*The key size to k_hmac_sha_1 in byte*/
int ld_hmac_sha_1; /*the length of data input in byte*/
int t_hmac_sha_1; /*The size of output from HMAC in byte*/

lk_hmac_sha_1=20;
t_hmac_sha_1=20;
max=200;

strcpy(challengeString,"01234567890123456789");
len1=big_to_bytes(max,signatureToTheCertificate_r,buffer1,FALSE);
buffer1[len1]='\0';
len2=big_to_bytes(max,signatureToTheCertificate_s,buffer2,FALSE);
buffer2[len2]='\0';
len3=big_to_bytes(max,sigMasterHashChainIdTimeStartStop_r,buffer3,FALSE);
buffer3[len3]='\0';
len4=big_to_bytes(max,sigMasterHashChainIdTimeStartStop_s,buffer4,FALSE);
buffer4[len4]='\0';
strcpy(message_1,owenAllCertificateElementThatIsSigned);
strcat(message_1,buffer1);
strcat(message_1,buffer2);
strcat(message_1,masterHashChainIdTimeStartStop);
strcat(message_1,buffer3);
strcat(message_1,buffer4);
strcat(message_1,challengeString);
strcat(message_1,kMasterLastUsed);
strcat(message_1,kTrafficLastUsed);
strcpy(buffer5,challengeString);
strcat(buffer5,kTrafficLastUsed);
```

```
for (ld_hmac_sha_1=0; buffer5[ld_hmac_sha_1]!='\0';ld_hmac_sha_1++);
hmac_sha_1(kMasterNext,&lk_hmac_sha_1,buffer5,&ld_hmac_sha_1,buffer6,
&t_hmac_sha_1);
strcat(message_1,buffer6);
}


void message2_ecdsa_dsa(char owenAllCertificateElementThatIsSigned_ecdsa[],
big signatureToTheCertificate_r,
big signatureToTheCertificate_s,
char masterHashChainIdTimeStartStop[],
big sigMasterHashChainIdTimeStartStop_r,
big sigMasterHashChainIdTimeStartStop_s,
char challengeStringToInitiator[],
int challengeToInitiatorInt,
char kMasterLastUsed[],
char kTrafficLastUsed[],
char kMasterNext[],
char macToChallengekMasterkTraffic[],
char message_2[],
char challengeFromInitiator[],
char hashOfchallengeToInitiator[])
{
int len1,len2,len3,len4,max,l;
char buffer1[200],buffer2[200],buffer3[200],buffer4[200];
char buffer5[200],buffer6[SHA_1_DIGESTSIZE+1];
int lk_hmac_sha_1; /*The key size to k_hmac_sha_1 in byte*/
int ld_hmac_sha_1; /*the length of data input in byte*/
int t_hmac_sha_1; /*The size of output from HMAC in byte*/

lk_hmac_sha_1=20;
t_hmac_sha_1=20;
max=200;
l=0;

len1=big_to_bytes(max,signatureToTheCertificate_r,buffer1,FALSE);
buffer1[len1]='\0';
len2=big_to_bytes(max,signatureToTheCertificate_s,buffer2,FALSE);
buffer2[len2]='\0';
len3=big_to_bytes(max,sigMasterHashChainIdTimeStartStop_r,buffer3,FALSE);
buffer3[len3]='\0';
len4=big_to_bytes(max,sigMasterHashChainIdTimeStartStop_s,buffer4,FALSE);
buffer4[len4]='\0';
strcpy(message_2,owenAllCertificateElementThatIsSigned_ecdsa);
strcat(message_2,buffer1);
strcat(message_2,buffer2);
```

```
strcat(message_2,masterHashChainIdTimeStartStop);
strcat(message_2,buffer3);
strcat(message_2,buffer4);

/*Establish the challenge to initiator and hash it*/
strcpy(challengeStringToInitiator,"9");
strcat(challengeStringToInitiator,challengeFromInitiator);
hashing_sha_1(challengeStringToInitiator,&l,hashOfchallengeToInitiator);

/*include the challenge in the message*/
strcat(message_2,hashOfchallengeToInitiator);
strcat(message_2,kMasterLastUsed);
strcat(message_2,kTrafficLastUsed);

/*generate MAC*/
strcpy(buffer5,hashOfchallengeToInitiator);
strcat(buffer5,kTrafficLastUsed);
for (ld_hmac_sha_1=0; buffer5[ld_hmac_sha_1]!='\0';ld_hmac_sha_1++);
hmac_sha_1(kMasterNext,&lk_hmac_sha_1,buffer5,&ld_hmac_sha_1,buffer6,
&t_hmac_sha_1);

strcat(message_2,buffer6);
}

void message3( int nodeID,
int otherNodeID,
char challengeFromResponder[],
char owenchallenge[],
char respondsTochallenge[],
char kTrafficNext[],
char mac[],
char message_3[])
{
char buffer1[SHA_1_DIGESTSIZE+1];
char buffer1_1[SHA_1_DIGESTSIZE+1];
char buffer2[SHA_1_DIGESTSIZE+1];
char buffer3[SHA_1_DIGESTSIZE+1];
int l,i,k,len1,len2;
int lk_hmac_sha_1; /*The key size to k_hmac_sha_1 in byte*/
int ld_hmac_sha_1; /*the length of data input in byte*/
int max,t_hmac_sha_1; /*The size of output from HMAC in byte*/

char stack_mem[mr_big_reserve(1,50)];
    big temp;
    memset(stack_mem,0,mr_big_reserve(1,50));
    temp=mirvar_mem(stack_mem,0);
```

```
lk_hmac_sha_1=20;
t_hmac_sha_1=20;
len1=SHA_1_DIGESTSIZE+1;
len2=0;
convert(len2,temp);

l=0;
k=0;
max=SHA_1_DIGESTSIZE+1;

for (i=0;i<(SHA_1_DIGESTSIZE+1);i++)
buffer3[i]='0';

_itoa(nodeID,buffer1,16);
_itoa(otherNodeID,buffer1_1,16);
for (;((len2<SHA_1_DIGESTSIZE)&&(0!=strcmp(buffer3,challengeFromResponder)));)
{
len2=big_to_bytes(max,temp,buffer2,FALSE);
buffer2[len2]='\0';
strcat(buffer2,owenchallenge);
hashing_sha_1(buffer2,&l,buffer3);
incr(temp,1,temp);
}
hashing_sha_1(buffer3,&l,respondsTochallenge);
strcpy(message_3,buffer1);
strcat(message_3,buffer1_1);
strcat(message_3,respondsTochallenge);
for (ld_hmac_sha_1=0; message_3[ld_hmac_sha_1]!='\0';ld_hmac_sha_1++);
hmac_sha_1(kTrafficNext,&lk_hmac_sha_1,message_3,&ld_hmac_sha_1,mac,
&t_hmac_sha_1);
strcat(message_3,mac);
}

void message4And8( int nodeID,
char kTrafficNext[],
char message_4_8[])
{
char buffer1[20];

_itoa(nodeID,buffer1,16);
strcpy(message_4_8,buffer1);
strcat(message_4_8,kTrafficNext);
}

void message5And9And11And13( int nodeID,
```

```
char kTrafficCurrent[],
char kMasterCurrent[],
char kMasterNext[],
char mac[],
char message_5_9_11_13[])
{
char buffer1[20],buffer2[50];
int lk_hmac_sha_1; /*The key size to k_hmac_sha_1 in byte*/
int ld_hmac_sha_1; /*the length of data input in byte*/
int i,t_hmac_sha_1; /*The size of output from HMAC in byte*/

lk_hmac_sha_1=20;
t_hmac_sha_1=20;

_itoa(nodeID,buffer1,16);
strcpy(message_5_9_11_13,buffer1);
strcat(message_5_9_11_13,kMasterCurrent);
strcat(message_5_9_11_13,kTrafficCurrent);

strcpy(buffer2,buffer1);
strcat(buffer2,kTrafficCurrent);

for (ld_hmac_sha_1=0; buffer2[ld_hmac_sha_1]!='\0';ld_hmac_sha_1++);
hmac_sha_1(kMasterNext,&lk_hmac_sha_1,buffer2,&ld_hmac_sha_1,mac,
&t_hmac_sha_1);
strcat(message_5_9_11_13,mac);
}

void message6And10And14(int nodeID,
char kTrafficCurrent[],
char kMasterCurrent[],
char kMasterNext[],
char mac[],
char message_6_10_14[])
{
char buffer1[20],buffer2[50];
int lk_hmac_sha_1; /*The key size to k_hmac_sha_1 in byte*/
int ld_hmac_sha_1; /*the length of data input in byte*/
int t_hmac_sha_1; /*The size of output from HMAC in byte*/

lk_hmac_sha_1=20;
t_hmac_sha_1=20;

_itoa(nodeID,buffer1,16);
strcpy(message_6_10_14,buffer1);
strcat(message_6_10_14,kMasterCurrent);
```

```
strcat(message_6_10_14,kTrafficCurrent);

strcpy(buffer2,buffer1);
strcat(buffer2,kTrafficCurrent);

for (ld_hmac_sha_1=0; buffer2[ld_hmac_sha_1]!='\0';ld_hmac_sha_1++);
hmac_sha_1(kMasterNext,&lk_hmac_sha_1,buffer2,&ld_hmac_sha_1,mac,
&t_hmac_sha_1);
strcat(message_6_10_14,mac);
}
void message7(int nodeID,
char messageToBeTransmited[],
char kTrafficNext[],
char mac[],
char message_7[])
{
char buffer1[SHA_1_DIGESTSIZE+1];

int l;
int lk_hmac_sha_1; /*The key size to k_hmac_sha_1 in byte*/
int ld_hmac_sha_1; /*the length of data input in byte*/
int t_hmac_sha_1; /*The size of output from HMAC in byte*/

lk_hmac_sha_1=20;
t_hmac_sha_1=20;

l=0;

_itoa(nodeID,buffer1,16);
strcpy(message_7,buffer1);
strcat(message_7,messageToBeTransmited);
for (ld_hmac_sha_1=0; message_7[ld_hmac_sha_1]!='\0';ld_hmac_sha_1++);
hmac_sha_1(kTrafficNext,&lk_hmac_sha_1,message_7,&ld_hmac_sha_1,mac,
&t_hmac_sha_1);
strcat(message_7,mac);
}
void message12( int nodeID,
char kTrafficCurrent[],
char kMasterNext[],
char mac[],
char message_12[])
{
char buffer1[SHA_1_DIGESTSIZE+1];
int l;
int lk_hmac_sha_1; /*The key size to k_hmac_sha_1 in byte*/
int ld_hmac_sha_1; /*the length of data input in byte*/
```

```
int t_hmac_sha_1; /*The size of output from HMAC in byte*/

lk_hmac_sha_1=20;
t_hmac_sha_1=20;

l=0;

_itoa(nodeID,buffer1,16);
strcpy(message_12,buffer1);
strcat(message_12,kTrafficCurrent);
for (ld_hmac_sha_1=0; message_12[ld_hmac_sha_1]!='\0';ld_hmac_sha_1++);
hmac_sha_1(kMasterNext,&lk_hmac_sha_1,message_12,&ld_hmac_sha_1,mac,
&t_hmac_sha_1);
strcat(message_12,mac);
}

int main()
{
big hash,hash_of_message;
big e_rsa; /*To be used within RSA*/
long seed; /*seed for DSA*/
long seed_RSA[2*NP]; /*seed for RSA*/
int c,i,j,l,ii,val,temp; /*counter and temp memory*/
long temp_rsa_e;
int ep_ecdsa; /*is used to compress point*/
long long int hash_time;

/*Node*/
struct node node_array[NRNODE];

/*Timing and clock*/
clock_t start, finish;
double duration;
time_t ltime;
long int hash_key_time0;

/*Hash chain*/
/*The h0 to be used to generate hash chain*/
char hash_seed[SHA_1_DIGESTSIZE+1]="01234567890123456789";

/*hash chain to be used with SHA-1*/
char temp_array[length_HASH_CHAIN][SHA_1_DIGESTSIZE+1];
char hashchain[length_HASH_CHAIN][SHA_1_DIGESTSIZE+1];

int l_hash_chain; /*The length of hash chain*/
char buffer1[120],buffer2[120],buffer3[120],buffer4[120];
```

```
/*HMAC and SHA*/
/*sha-1*/
/*The key that is used within HMAC*/
char *pk_hmac_sha_1,k_hmac_sha_1[SHA_1_DIGESTSIZE+1]="Jefe";

/*The key size to k_hmac_sha_1 in byte*/
int lk_hmac_sha_1;

/*Data that shall be authenticated,
used as test parameter from RFC 2202 to test the implementation of HMAC*/
char *pd_hmac_sha_1,d_hmac_sha_1[80]="what do ya want for nothing?";

int ld_hmac_sha_1; /*the length of data input in byte*/
char *pout_hmac_sha_1,out_hmac_sha_1[SHA_1_DIGESTSIZE+1]; /*The HMAC output*/
int t_hmac_sha_1;  /*The size of output from HMAC in byte*/

mip=mirsys(256,16);

    hash=mirvar(0);
hash_of_message=mirvar(0);

/*definition of variable for RSA*/
e_rsa=mirvar(0);

for (i=0; i<NRNODE; i++)
{
/*The curve variable init*/
node_array[i].owen_certificate.curve_ecdsa.a=mirvar(0);
node_array[i].owen_certificate.curve_ecdsa.b=mirvar(0);
node_array[i].owen_certificate.curve_ecdsa.p=mirvar(0);
node_array[i].owen_certificate.curve_ecdsa.q=mirvar(0);
node_array[i].owen_certificate.curve_ecdsa.x=mirvar(0);
node_array[i].owen_certificate.curve_ecdsa.y=mirvar(0);

node_array[i].others_certificate.curve_ecdsa.a=mirvar(0);
node_array[i].others_certificate.curve_ecdsa.b=mirvar(0);
node_array[i].others_certificate.curve_ecdsa.p=mirvar(0);
node_array[i].others_certificate.curve_ecdsa.q=mirvar(0);
node_array[i].others_certificate.curve_ecdsa.x=mirvar(0);
node_array[i].others_certificate.curve_ecdsa.y=mirvar(0);

node_array[i].ca_certificate.curve_ecdsa.a=mirvar(0);
node_array[i].ca_certificate.curve_ecdsa.b=mirvar(0);
node_array[i].ca_certificate.curve_ecdsa.p=mirvar(0);
node_array[i].ca_certificate.curve_ecdsa.q=mirvar(0);
```

```
node_array[i].ca_certificate.curve_ecdsa.x=mirvar(0);
node_array[i].ca_certificate.curve_ecdsa.y=mirvar(0);

/*The certificate initalisation of variable*/
/*rsa*/
node_array[i].owen_certificate.rsa.SubjectPublicKeyInfo.public_key_rsa.n=
mirvar(0);
node_array[i].owen_certificate.rsa.SubjectPublicKeyInfo.public_key_rsa.e=
mirvar(0);
node_array[i].owen_certificate.rsa.sign_certificate_rsa=mirvar(0);
node_array[i].owen_certificate.rsa.hash_certificate=mirvar(0);

node_array[i].others_certificate.rsa.SubjectPublicKeyInfo.public_key_rsa.n=
mirvar(0);
node_array[i].others_certificate.rsa.SubjectPublicKeyInfo.public_key_rsa.e=
mirvar(0);
node_array[i].others_certificate.rsa.sign_certificate_rsa=mirvar(0);

node_array[i].ca_certificate.rsa.SubjectPublicKeyInfo.public_key_rsa.n=
mirvar(0);
node_array[i].ca_certificate.rsa.SubjectPublicKeyInfo.public_key_rsa.e=
mirvar(0);
node_array[i].ca_certificate.rsa.sign_certificate_rsa=mirvar(0);

/*dsa*/
node_array[i].owen_certificate.dsa.SubjectPublicKeyInfo.public_key_dsa.g=
mirvar(0);
node_array[i].owen_certificate.dsa.SubjectPublicKeyInfo.public_key_dsa.k=
mirvar(0);
node_array[i].owen_certificate.dsa.SubjectPublicKeyInfo.public_key_dsa.p=
mirvar(0);
node_array[i].owen_certificate.dsa.SubjectPublicKeyInfo.public_key_dsa.q=
mirvar(0);
node_array[i].owen_certificate.dsa.SubjectPublicKeyInfo.public_key_dsa.y=
mirvar(0);
node_array[i].owen_certificate.dsa.sign_certificate_dsa.r=mirvar(0);
node_array[i].owen_certificate.dsa.sign_certificate_dsa.s=mirvar(0);
node_array[i].owen_certificate.dsa.hash_certificate=mirvar(0);

node_array[i].others_certificate.dsa.SubjectPublicKeyInfo.public_key_dsa.g=
mirvar(0);
node_array[i].others_certificate.dsa.SubjectPublicKeyInfo.public_key_dsa.k=
mirvar(0);
node_array[i].others_certificate.dsa.SubjectPublicKeyInfo.public_key_dsa.p=
mirvar(0);
node_array[i].others_certificate.dsa.SubjectPublicKeyInfo.public_key_dsa.q=
```

```
mirvar(0);
node_array[i].others_certificate.dsa.SubjectPublicKeyInfo.public_key_dsa.y=
mirvar(0);
node_array[i].others_certificate.dsa.sign_certificate_dsa.r=mirvar(0);
node_array[i].others_certificate.dsa.sign_certificate_dsa.s=mirvar(0);

node_array[i].ca_certificate.dsa.SubjectPublicKeyInfo.public_key_dsa.g=
mirvar(0);
node_array[i].ca_certificate.dsa.SubjectPublicKeyInfo.public_key_dsa.k=
mirvar(0);
node_array[i].ca_certificate.dsa.SubjectPublicKeyInfo.public_key_dsa.p=
mirvar(0);
node_array[i].ca_certificate.dsa.SubjectPublicKeyInfo.public_key_dsa.q=
mirvar(0);
node_array[i].ca_certificate.dsa.SubjectPublicKeyInfo.public_key_dsa.y=
mirvar(0);
node_array[i].ca_certificate.dsa.sign_certificate_dsa.r=mirvar(0);
node_array[i].ca_certificate.dsa.sign_certificate_dsa.s=mirvar(0);

/*ecdsa*/
node_array[i].owen_certificate.ecdsa.SubjectPublicKeyInfo.
public_key_ecdsa.x=mirvar(0);
node_array[i].owen_certificate.ecdsa.sign_certificate_ecdsa.r=mirvar(0);
node_array[i].owen_certificate.ecdsa.sign_certificate_ecdsa.s=mirvar(0);
node_array[i].owen_certificate.ecdsa.hash_certificate=mirvar(0);

node_array[i].others_certificate.ecdsa.SubjectPublicKeyInfo.
public_key_ecdsa.x=mirvar(0);
node_array[i].others_certificate.ecdsa.sign_certificate_ecdsa.r=
mirvar(0);
node_array[i].others_certificate.ecdsa.sign_certificate_ecdsa.s=
mirvar(0);
node_array[i].ca_certificate.ecdsa.SubjectPublicKeyInfo.public_key_ecdsa.x=
mirvar(0);
node_array[i].ca_certificate.ecdsa.sign_certificate_ecdsa.r=mirvar(0);
node_array[i].ca_certificate.ecdsa.sign_certificate_ecdsa.s=mirvar(0);

/*The private key variable init*/
/*RSA*/
node_array[i].owen_private_key.rsa.d=mirvar(0);
node_array[i].owen_private_key.rsa.p=mirvar(0);
node_array[i].owen_private_key.rsa.q=mirvar(0);

/*DSA*/
node_array[i].owen_private_key.dsa.x=mirvar(0);
```

```
/*ECDSA*/
node_array[i].owen_private_key.ecdsa.d=mirvar(0);

/*Variable used to trans./receive etc*/

node_array[i].transmit.hash=mirvar(0);
node_array[i].transmit.sign=mirvar(0);
node_array[i].transmit.verification=mirvar(0);
node_array[i].transmit.encryption=mirvar(0);
node_array[i].transmit.decryption=mirvar(0);

node_array[i].receive.hash=mirvar(0);
node_array[i].receive.sign=mirvar(0);
node_array[i].receive.verification=mirvar(0);
node_array[i].receive.encryption=mirvar(0);
node_array[i].receive.decryption=mirvar(0);

node_array[i].hash_to_id_last_master_hash_time0_lifetime_big=mirvar(0);
node_array[i].sign_master_hash_chain.sign_ecdsa.r=mirvar(0);
node_array[i].sign_master_hash_chain.sign_ecdsa.s=mirvar(0);
node_array[i].sign_master_hash_chain.sign_rsa.s=mirvar(0);
node_array[i].sign_master_hash_chain.sign_dsa.r=mirvar(0);
node_array[i].sign_master_hash_chain.sign_dsa.s=mirvar(0);
}


/*initialisation of seed in DSA,RSA,ECDSA*/
seed=123456789; /*to bee used in DSA,RSA,ECDSA*/

temp_rsa_e=65537; /*to be used as private e key in RSA*/
for (i=0; i<NRNODE; i++)
{
node_array[i].owen_private_key.dsa.seed=seed;
node_array[i].owen_private_key.rsa.seed=seed;
for (j=0; j<NP; j++)
node_array[i].owen_private_key.rsa.seed_RSA[j]=seed;
node_array[i].owen_private_key.ecdsa.seed=seed;
}


/*initialisation of hash of a message*/
temp=987654321;

/*convert an integer temp to a big hash_of_message, to be used in RSA and DSA*/
convert(temp,hash_of_message);
val=3; /*initialisation of validation parameter, to be used in DSA*/
```

```
i=0; /*initialisation, counter set to zero*/

/* convert from integer to big initialisation of e_rsa(big)=
temp_rsa_e(int), to be used in RSA*/
convert(temp_rsa_e,e_rsa);

/*initialisation of hash chain*/
l_hash_chain=length_HASH_CHAIN;

mip->IOBASE=16;
for (i=0; i<NRNODE; i++)
{
node_array[i].owen_private_key.ecdsa.seed=seed;
convert(a_secp_160,node_array[i].owen_certificate.curve_ecdsa.a);
cinstr(node_array[i].owen_certificate.curve_ecdsa.b,b_secp_160);
cinstr(node_array[i].owen_certificate.curve_ecdsa.p,p_secp_160);
cinstr(node_array[i].owen_certificate.curve_ecdsa.q,q_secp_160);
cinstr(node_array[i].owen_certificate.curve_ecdsa.x,x_secp_160);
cinstr(node_array[i].owen_certificate.curve_ecdsa.y,y_secp_160);
}

/*Generate hash chain for nodes*/
printf("\n---------------Generate hash chain------------\n");
printf("hash_seed=");
for (i=0; hash_seed[i]!='\0';/*i<SHA_1_DIGESTSIZE;*/ i++)
printf("%x",(unsigned char)hash_seed[i]);
printf("\n");


/*Master hash chain*/
printf("\n------------Generate master hash chain-------------------\n");
l_hash_chain=MAX_length_MASTER_HASH_CHAIN;
start=clock();
for (j=0; j<NR2; j++)
{
for (i=0; i<NRNODE; i++)
{
hash_chain_sha_1( hash_seed,
&l_hash_chain,
node_array[i].Master_hash_chain);
time(&ltime);
node_array[i].master_hash_key_time0=ltime;
node_array[i].master_hash_key_end_life_time=node_array[i].
master_hash_key_time0+MAX_length_MASTER_HASH_CHAIN;
}
}
```

```
finish=clock();
duration=(double)(finish-start)/CLOCKS_PER_SEC;
printf("\n");
printf("Number of interaction NR=%d\n",j);
printf("Number of Node=%d\n",i);
printf("The length of hash chain=%d\n",l_hash_chain);
printf("Time used to generate master hash chain in s=%f\n",duration);

start=clock();
for (i=0; i<NRNODE; i++)
hash_laste_master_hash_chain_key(
node_array[i].owen_certificate.ecdsa.CertificateSerialNumber,
node_array[i].Master_hash_chain[MAX_length_MASTER_HASH_CHAIN-1],
node_array[i].master_hash_key_time0,
node_array[i].master_hash_key_end_life_time,
node_array[i].id_last_master_hash_time0_lifetime_in_array,
node_array[i].hash_to_id_last_master_hash_time0_lifetime_big);

finish=clock();
duration=(double)(finish-start)/CLOCKS_PER_SEC;
printf("Time used to generate master hash chain key in ms=%f\n",
duration*1000);

/*Traffic hash chain*/
printf("\n------------Generate traffic hash chain-------------------\n");
l_hash_chain=MAX_length_TRAFFIC_HASH_CHAIN;
start=clock();
for (i=0; i<NRNODE; i++)
{
hash_chain_sha_1( hash_seed,
&l_hash_chain,
node_array[i].Traffic_hash_chain);
node_array[i].index_kTraffic=0;
}

finish=clock();
duration=(double)(finish-start)/CLOCKS_PER_SEC;
printf("\n");
printf("Number of interaction (NR of Node)=%d\n",i);
printf("The length of hash chain=%d\n",l_hash_chain);
printf("generate traffic hash chain in ms=%f\n",duration*1000);

/*Session hash chain*/
printf("\n------------Generate session hash chain-------------------\n");
l_hash_chain=MAX_length_OF_SESSION_HASH_CHAIN;
start=clock();
```

```
for (i=0; i<NRNODE; i++)
for (j=0; j<MAX_NR_OF_SESSION; j++)
{
hash_chain_sha_1( hash_seed,
&l_hash_chain,
node_array[i].session_hash_chain[j]);
node_array[i].index_session_key[j]=0;
}
finish=clock();
duration=(double)(finish-start)/CLOCKS_PER_SEC;
printf("\n");
printf("Number of interaction (NR of Node)=%d\n",i);
printf("The number of session hash chain=%d\n",j);
printf("The length of hash chain=%d\n",l_hash_chain);
printf("Time used to generate session hash chain in ms=%f\n",duration*1000);
printf("\n");

/*DSS setup*/
printf("-----------DSS setup---------------\n");
start=clock();
for (j=0; j<NR2; j++)
for (i=0; i<NRNODE; i++)
dss_setup( node_array[i].owen_certificate.dsa.SubjectPublicKeyInfo.
public_key_dsa.p,
node_array[i].owen_certificate.dsa.SubjectPublicKeyInfo.
public_key_dsa.q,
node_array[i].owen_certificate.dsa.SubjectPublicKeyInfo.
public_key_dsa.g,
node_array[i].owen_private_key.dsa.seed);
finish=clock();
duration=(double)(finish-start)/CLOCKS_PER_SEC;
printf("\n");
printf("Number of interaction j=1%d\n",j);
printf("Number of interaction i=1%d\n",i);
printf("elapsed time is in ms:%f\n",duration*1000);
printf("\n");

/* for (i=0; i<NRNODE; i++)
{
printf("\nFor node nr: %d\n",i);
printf("The seed is= %ld\n", node_array[i].owen_private_key.dsa.seed);
printf("q= ");
cotnum(node_array[i].owen_certificate.dsa.SubjectPublicKeyInfo.
public_key_dsa.q,stdout);
printf("p= ");
cotnum(node_array[i].owen_certificate.dsa.SubjectPublicKeyInfo.
```

```
public_key_dsa.p,stdout);
printf("g= ");
cotnum(node_array[i].owen_certificate.dsa.SubjectPublicKeyInfo.
public_key_dsa.g,stdout);
}
*/
/*DSS generation*/
printf("-----------DSS generation---------------\n");
start=clock();
for (j=0; j<NR/10; j++)
for (i=0; i<NRNODE; i++)
dss_gen( node_array[i].owen_certificate.dsa.SubjectPublicKeyInfo.public_key_dsa.p,
node_array[i].owen_certificate.dsa.SubjectPublicKeyInfo.public_key_dsa.q,
node_array[i].owen_certificate.dsa.SubjectPublicKeyInfo.public_key_dsa.g,
node_array[i].owen_private_key.dsa.x,
node_array[i].owen_certificate.dsa.SubjectPublicKeyInfo.public_key_dsa.y,
node_array[i].owen_private_key.dsa.seed);
finish=clock();
duration=(double)(finish-start)/CLOCKS_PER_SEC;
printf("\n");
printf("Number of interaction j=%d\n",j);
printf("Number of interaction i=%d\n",i);
printf("Elapsed time is in ms:%f\n",duration*1000);
printf("\n");

/* for (i=0; i<NRNODE; i++)
{
printf("\nFor node nr: %d\n",i);
printf("The seed is= %ld\n", node_array[i].owen_private_key.dsa.seed);
printf("q= ");
cotnum(node_array[i].owen_certificate.dsa.SubjectPublicKeyInfo.
public_key_dsa.q,stdout);
printf("p= ");
cotnum(node_array[i].owen_certificate.dsa.SubjectPublicKeyInfo.
public_key_dsa.p,stdout);
printf("g= ");
cotnum(node_array[i].owen_certificate.dsa.SubjectPublicKeyInfo.
public_key_dsa.g,stdout);
printf("public key y = ");
otnum(node_array[i].owen_certificate.dsa.SubjectPublicKeyInfo.
public_key_dsa.y,stdout);
printf("private key x = ");
otnum(node_array[i].owen_private_key.dsa.x,stdout);
}
*/
for (i=0; i<NRNODE; i++)
```

```
{
time(&ltime); /*The UTC time that will be used in the certificate*/
node_array[i].owen_certificate.dsa.CertificateSerialNumber=20000+i;
node_array[i].owen_certificate.dsa.Time.generalizedTime=ltime;
node_array[i].owen_certificate.dsa.Time.utcTime=ltime;
node_array[i].owen_certificate.dsa.Validity.notBefore=ltime;
node_array[i].owen_certificate.dsa.Validity.notAfter=ltime+100000000;
node_array[i].owen_certificate.dsa.AlgorithmIdentifier.ALGORITHM_id=2;
strcpy(node_array[i].owen_certificate.dsa.AlgorithmIdentifier.
ALGORITHM_Type,"DSA");
strcpy(node_array[i].owen_certificate.dsa.SupportedAlgorithms,
"DSASHA1");
node_array[i].owen_certificate.dsa.version=0;
strcpy(node_array[i].owen_certificate.dsa.issuer,"CA");
node_array[i].owen_certificate.dsa.subject=node_array[i].
owen_certificate.dsa.CertificateSerialNumber;
strcpy(node_array[i].owen_certificate.dsa.SubjectPublicKeyInfo.
algorithm,"DSA");
}

/*Signature on DSA certificate*/
start=clock();
for (j=0; j<NR; j++)
for (i=0; i<NRNODE; i++)
{
hash_of_dsa_certificate(
node_array[i].owen_certificate.dsa.version,
node_array[i].owen_certificate.dsa.CertificateSerialNumber,
node_array[i].owen_certificate.dsa.AlgorithmIdentifier.
ALGORITHM_id,
node_array[i].owen_certificate.dsa.AlgorithmIdentifier.
ALGORITHM_Type,
node_array[i].owen_certificate.dsa.issuer,
node_array[i].owen_certificate.dsa.Validity.notBefore,
node_array[i].owen_certificate.dsa.Validity.notAfter,
node_array[i].owen_certificate.dsa.subject,
node_array[i].owen_certificate.dsa.SubjectPublicKeyInfo.
algorithm,
node_array[i].owen_certificate.dsa.SubjectPublicKeyInfo.
public_key_dsa.g,
node_array[i].owen_certificate.dsa.SubjectPublicKeyInfo.
public_key_dsa.k,
node_array[i].owen_certificate.dsa.SubjectPublicKeyInfo.
public_key_dsa.p,
node_array[i].owen_certificate.dsa.SubjectPublicKeyInfo.
public_key_dsa.q,
```

```
node_array[i].owen_certificate.dsa.SubjectPublicKeyInfo.
public_key_dsa.y,
node_array[i].owen_certificate.dsa.hash_certificate,
node_array[i].owen_certificate.dsa.
AllCertificateElementThatIsSigned);
}
finish=clock();
duration=(double)(finish-start)/CLOCKS_PER_SEC;
printf("\n");
printf("Number of interaction j=%d\n",j);
printf("Number of interaction i=%d\n",i);
printf("Elapsed time is in ms:%f\n",duration*1000);
printf("\n");


/*DSS signature on certificate*/
printf("-----------DSS signature--------------\n");
/*Node "0" is the CA*/
start=clock();
for (j=0; j<NR; j++)
for (i=0; i<NRNODE; i++)
dss_sign(
node_array[0].owen_certificate.dsa.SubjectPublicKeyInfo.
public_key_dsa.p,
node_array[0].owen_certificate.dsa.SubjectPublicKeyInfo.
public_key_dsa.q,
node_array[0].owen_certificate.dsa.SubjectPublicKeyInfo.
public_key_dsa.g,
node_array[0].owen_private_key.dsa.x,
node_array[i].owen_certificate.dsa.hash_certificate,
node_array[i].owen_certificate.dsa.sign_certificate_dsa.r,
node_array[i].owen_certificate.dsa.sign_certificate_dsa.s,
node_array[i].owen_private_key.dsa.seed);
finish=clock();
duration=(double)(finish-start)/CLOCKS_PER_SEC;
printf("\n");
printf("Number of interaction j=%d\n",j);
printf("Number of interaction i=%d\n",i);
printf("Elapsed time is in ms:%f\n",duration*1000);
printf("\n");
for (i=0; i<NRNODE; i++)
for (j=0; node_array[i].owen_certificate.dsa.
AllCertificateElementThatIsSigned[j]!='\0';j++);
printf("\n The lengt to DSA certificate is=%d\n",j);
```

```
/* printf("\n CA certificate signature_s=\n");
cotnum(node_array[0].owen_certificate.dsa.sign_certificate_dsa.s,stdout);
*/ for (i=0; i<NRNODE; i++)
{
/* printf("\nFor node nr: %d\n",i);*/
node_array[i].ca_certificate.dsa=node_array[0].owen_certificate.dsa;
/* printf("\n Node n CA certificate signature_s=%d\n",i);
cotnum(node_array[i].ca_certificate.dsa.
sign_certificate_dsa.s,stdout);*/
}

mip->IOBASE=16;
/* for (i=0; i<NRNODE; i++)
{
printf("\nFor node nr: %d\n",i);
printf("\nDSA parameters for Node=%d\n",i);
printf("new private key x= \n");
cotnum(node_array[i].owen_private_key.dsa.x,stdout);
printf("seed= %d\n",node_array[i].owen_private_key.dsa.seed);
printf("public key (g)=\n");
cotnum(node_array[i].owen_certificate.dsa.SubjectPublicKeyInfo.
public_key_dsa.g,stdout);
printf("public key (k)=\n");
cotnum(node_array[i].owen_certificate.dsa.SubjectPublicKeyInfo.
public_key_dsa.k,stdout);
printf("public key (p)=\n");
cotnum(node_array[i].owen_certificate.dsa.SubjectPublicKeyInfo.
public_key_dsa.p,stdout);
printf("public key (q)=\n");
cotnum(node_array[i].owen_certificate.dsa.SubjectPublicKeyInfo.
public_key_dsa.q,stdout);
printf("public key (y)=\n");
cotnum(node_array[i].owen_certificate.dsa.SubjectPublicKeyInfo.
public_key_dsa.y,stdout);
printf("owenAllCertificateElementThatIsSigned_dsa=\n");
for (j=0; node_array[i].owen_certificate.dsa.
AllCertificateElementThatIsSigned[j]!='\0';j++)
printf("%x",(unsigned char)node_array[i].owen_certificate.dsa.
AllCertificateElementThatIsSigned[j]);
printf("\n The certificate length in char is=%d\n",j);
printf("\nsign_certificate_dsa.r=\n");
cotnum(node_array[i].owen_certificate.dsa.sign_certificate_dsa.
r,stdout);
printf("sign_certificate_dsa.s=\n");
cotnum(node_array[i].owen_certificate.dsa.sign_certificate_dsa.
s,stdout);
```

```
}
*/

/*DSS verification*/
printf("\n-----------DSS verification of certificate---------------\n");
start=clock();
for (j=0; j<NR; j++)
for (i=0; i<NRNODE; i++)
dss_ver( node_array[i].ca_certificate.dsa.SubjectPublicKeyInfo.
public_key_dsa.p,
node_array[i].ca_certificate.dsa.SubjectPublicKeyInfo.
public_key_dsa.q,
node_array[i].ca_certificate.dsa.SubjectPublicKeyInfo.
public_key_dsa.g,
node_array[i].ca_certificate.dsa.SubjectPublicKeyInfo.
public_key_dsa.y,
node_array[i].owen_certificate.dsa.sign_certificate_dsa.r,
node_array[i].owen_certificate.dsa.sign_certificate_dsa.s,
node_array[i].owen_certificate.dsa.hash_certificate,
&node_array[i].val);
finish=clock();
duration=(double)(finish-start)/CLOCKS_PER_SEC;
printf("\n");
printf("Number of interaction j=%d\n",j);
printf("Number of interaction i=%d\n",i);
printf("Elapsed time is in ms:%f\n",duration*1000);
printf("\n");

/* for (i=0; i<NRNODE; i++)
{
printf("\nFor node nr: %d\n",i);
printf("The seed is= %ld\n",node_array[i].owen_private_key.dsa.seed);
printf("q= ");
cotnum(node_array[i].owen_certificate.dsa.SubjectPublicKeyInfo.
public_key_dsa.q,stdout);
printf("p= ");
cotnum(node_array[i].owen_certificate.dsa.SubjectPublicKeyInfo.
public_key_dsa.p,stdout);
printf("g= ");
cotnum(node_array[i].owen_certificate.dsa.SubjectPublicKeyInfo.
public_key_dsa.g,stdout);
printf("private key x = ");
otnum(node_array[i].owen_private_key.dsa.x,stdout);
printf("hash_of_message = ");
otnum(node_array[i].owen_certificate.dsa.hash_certificate,stdout);
printf("Signature on certificate r = ");
```

```
otnum(node_array[i].owen_certificate.dsa.sign_certificate_dsa.r,stdout);
printf("Signature on certificate s = ");
otnum(node_array[i].owen_certificate.dsa.sign_certificate_dsa.s,stdout);
printf("validation = %d\n",node_array[i].val);
if (node_array[i].val==1)
printf("The DSS Signature is verified\n");
else printf("The DSS Signature is NOT verified the second test\n");
}

*/
/*DSS signature on first key in master hash chain*/
printf("-----------DSS signature on first master hash chain-------------\n");

start=clock();
for (j=0; j<NR; j++)
for (i=0; i<NRNODE; i++)
dss_sign(
node_array[i].owen_certificate.dsa.SubjectPublicKeyInfo.
public_key_dsa.p,
node_array[i].owen_certificate.dsa.SubjectPublicKeyInfo.
public_key_dsa.q,
node_array[i].owen_certificate.dsa.SubjectPublicKeyInfo.
public_key_dsa.g,
node_array[i].owen_private_key.dsa.x,
node_array[i].hash_to_id_last_master_hash_time0_lifetime_big,
node_array[i].sign_master_hash_chain.sign_dsa.r,
node_array[i].sign_master_hash_chain.sign_dsa.s,
node_array[i].owen_private_key.dsa.seed);
finish=clock();
duration=(double)(finish-start)/CLOCKS_PER_SEC;
printf("\n");
printf("Number of interaction j=%d\n",j);
printf("Number of interaction i=%d\n",i);
printf("Elapsed time is in ms:%f\n",duration*1000);
printf("\n");


/*RSA key generation*/
printf("-----------RSA key generation---------------\n");
start=clock();
for (j=0; j<NR2; j++)
for (i=0; i<NRNODE; i++)
rsa_keygen( &node_array[i].owen_private_key.rsa.seed,
node_array[i].owen_certificate.rsa.
SubjectPublicKeyInfo.public_key_rsa.n,
node_array[i].owen_private_key.rsa.p,
```

```
node_array[i].owen_private_key.rsa.q,
node_array[i].owen_certificate.rsa.
SubjectPublicKeyInfo.public_key_rsa.e,
node_array[i].owen_private_key.rsa.d);
finish=clock();
duration=(double)(finish-start)/CLOCKS_PER_SEC;
printf("\n");
printf("Number of interaction j=%d\n",j);
printf("Number of interaction i=%d\n",i);
printf("Elapsed time is in ms:%f\n",duration*1000);
printf("\n");

/* for (i=0; i<NRNODE; i++)
{
printf("\nFor node nr: %d\n",i);
printf("\n----Generating 512-bit=PRIME_BITS primes p and q----\n");
printf("key length to n= %d bits\n",logb2(node_array[i].
owen_certificate.rsa.SubjectPublicKeyInfo.public_key_rsa.n));
printf("public encryption key (e):\n");
cotnum(node_array[i].owen_certificate.rsa.SubjectPublicKeyInfo.
public_key_rsa.e,stdout);
printf("private encryption key (d):\n");
cotnum(node_array[i].owen_private_key.rsa.d,stdout);
printf("Public key (n):\n");
cotnum(node_array[i].owen_certificate.rsa.SubjectPublicKeyInfo.
public_key_rsa.n,stdout);
printf("Key (p):\n");
cotnum(node_array[i].owen_private_key.rsa.p,stdout);
printf("Key (q):\n");
cotnum(node_array[i].owen_private_key.rsa.q,stdout);
}
*/
/*rsa encryption*/
/*printf("------------rsa encryption-------------------\n");
rsa_encrypt(n_rsa,p_rsa,q_rsa,e_rsa,d_rsa,hash_of_message,rsa_encrypt_m);
printf("public encryption key (d):\n");
    cotnum(d_rsa,stdout);
printf("private encryption key (e):\n");
    cotnum(e_rsa,stdout);
printf("Public key (n):\n");
    cotnum(n_rsa,stdout);
printf("Key (p):\n");
    cotnum(p_rsa,stdout);
printf("Key (q):\n");
    cotnum(q_rsa,stdout);
printf("Plain text (hash)= \n");
```

```
    cotnum(hash_of_message,stdout);*/


/*rsa decryption*/
/*printf("------------rsa decryption-------------------\n");
rsa_decrypt(n_rsa,p_rsa,q_rsa,e_rsa,d_rsa,rsa_decrypt_m,rsa_encrypt_m);
printf("public encryption key (d):\n");
    cotnum(d_rsa,stdout);
printf("private encryption key (e):\n");
    cotnum(e_rsa,stdout);
printf("Public key (n):\n");
    cotnum(n_rsa,stdout);
printf("Key (p):\n");
    cotnum(p_rsa,stdout);
printf("Key (q):\n");
    cotnum(q_rsa,stdout);
printf("Encrypting test string=\n");
cotnum(rsa_encrypt_m,stdout);
printf("Plain text (hash)= \n");
    cotnum(rsa_decrypt_m,stdout);*/


for (i=0; i<NRNODE; i++)
{
time(&ltime); /*The UTC time that will be used in the certificate*/
node_array[i].owen_certificate.rsa.CertificateSerialNumber=10000+i;
node_array[i].owen_certificate.rsa.Time.generalizedTime=ltime;
node_array[i].owen_certificate.rsa.Time.utcTime=ltime;
node_array[i].owen_certificate.rsa.Validity.notBefore=ltime;
node_array[i].owen_certificate.rsa.Validity.notAfter=ltime+100000000;
node_array[i].owen_certificate.rsa.AlgorithmIdentifier.ALGORITHM_id=1;
strcpy(node_array[i].owen_certificate.rsa.AlgorithmIdentifier.
ALGORITHM_Type,"RSA");
strcpy(node_array[i].owen_certificate.rsa.
SupportedAlgorithms,"RSA,SHA1");
node_array[i].owen_certificate.rsa.version=0;
strcpy(node_array[i].owen_certificate.rsa.issuer,"CA");
node_array[i].owen_certificate.rsa.subject=node_array[i].
owen_certificate.rsa.CertificateSerialNumber;
strcpy(node_array[i].owen_certificate.rsa.
SubjectPublicKeyInfo.algorithm,"RSA");
}

/*hash of RSA certificate*/
printf("-----------Hash of RSA certificate---------------\n");
start=clock();
```

```
for (j=0; j<NR; j++)
for (i=0; i<NRNODE; i++)
hash_of_rsa_certificate(
node_array[i].owen_certificate.rsa.version,
node_array[i].owen_certificate.rsa.CertificateSerialNumber,
node_array[i].owen_certificate.rsa.AlgorithmIdentifier.
ALGORITHM_id,
node_array[i].owen_certificate.rsa.AlgorithmIdentifier.
ALGORITHM_Type,
node_array[i].owen_certificate.rsa.issuer,
node_array[i].owen_certificate.rsa.Validity.notBefore,
node_array[i].owen_certificate.rsa.Validity.notAfter,
node_array[i].owen_certificate.rsa.subject,
node_array[i].owen_certificate.rsa.SubjectPublicKeyInfo.
algorithm,
node_array[i].owen_certificate.rsa.SubjectPublicKeyInfo.
public_key_rsa.n,
node_array[i].owen_certificate.rsa.SubjectPublicKeyInfo.
public_key_rsa.e,
node_array[i].owen_certificate.rsa.hash_certificate,
node_array[i].owen_certificate.rsa.
AllCertificateElementThatIsSigned);
finish=clock();
duration=(double)(finish-start)/CLOCKS_PER_SEC;
printf("\n");
printf("Number of interaction j=%d\n",j);
printf("Number of interaction i=%d\n",i);
printf("Elapsed time is in ms:%f\n",duration*1000);
printf("\n");

/* for (i=0; i<NRNODE; i++)
{
printf("\nFor node nr: %d\n",i);
printf("owenAllCertificateElementThatIsSigned_rsa=\n");
for (j=0; node_array[i].owen_certificate.rsa.
AllCertificateElementThatIsSigned[j]!='\0';j++)
printf("%x",(unsigned char)node_array[i].owen_certificate.rsa.
AllCertificateElementThatIsSigned[j]);
printf("\n The certificate length in char is=%d\n",j);
printf("The hash to the certificate=\n");
cotnum(node_array[i].owen_certificate.rsa.hash_certificate,stdout);
}
*/
/*RSA signature generation on certificate*/
printf("\n-----------RSA signature generation on certificate---------\n");
```

```
/*Node "0" is the CA*/
start=clock();
for (j=0; j<NR; j++)
for (i=0; i<NRNODE; i++)
rsa_signature( &node_array[0].owen_certificate.rsa.
SubjectPublicKeyInfo.public_key_rsa.n,
&node_array[0].owen_private_key.rsa.p,
&node_array[0].owen_private_key.rsa.q,
&node_array[0].owen_certificate.rsa.
SubjectPublicKeyInfo.public_key_rsa.e,
&node_array[0].owen_private_key.rsa.d,
&node_array[i].owen_certificate.rsa.
sign_certificate_rsa,
&node_array[i].owen_certificate.rsa.
hash_certificate);
finish=clock();
duration=(double)(finish-start)/CLOCKS_PER_SEC;
printf("\n");
printf("Number of interaction j=%d\n",j);
printf("Number of interaction i=%d\n",i);
printf("Elapsed time is in ms:%f\n",duration*1000);
printf("\n");
for (i=0; i<NRNODE; i++)
for (j=0; node_array[i].owen_certificate.rsa.
AllCertificateElementThatIsSigned[j]!='\0';j++);
printf("\n The length to RSA certificate is=%d\n",j);

/* printf("\n CA certificate signature_s=\n");
cotnum(node_array[0].owen_certificate.rsa.sign_certificate_rsa,stdout);
*/ for (i=0; i<NRNODE; i++)
{
/*printf("\nFor node nr: %d\n",i);*/
node_array[i].ca_certificate.rsa=node_array[0].owen_certificate.rsa;
/*printf("\n Node n CA certificate signature=%d\n",i);
cotnum(node_array[i].ca_certificate.rsa.sign_certificate_rsa,stdout);*/
}

mip->IOBASE=16;
/* for (i=0; i<NRNODE; i++)
{
printf("\nFor node nr: %d\n",i);
printf("\nRSA parameters for Node=%d\n",i);
printf("private key p= \n");
cotnum(node_array[i].owen_private_key.rsa.p,stdout);
printf("private key q= \n");
cotnum(node_array[i].owen_private_key.rsa.q,stdout);
```

```
printf("private key d= \n");
cotnum(node_array[i].owen_private_key.rsa.d,stdout);
printf("seed= %d\n",node_array[i].owen_private_key.rsa.seed);
printf("public key (n)=\n");
cotnum(node_array[i].owen_certificate.rsa.SubjectPublicKeyInfo.
public_key_rsa.n,stdout);
printf("new public key (e)=\n");
cotnum(node_array[i].owen_certificate.rsa.SubjectPublicKeyInfo.
public_key_rsa.e,stdout);
printf("owenAllCertificateElementThatIsSigned_rsa=\n");
for (j=0; node_array[i].owen_certificate.rsa.
AllCertificateElementThatIsSigned[j]!='\0';j++)
printf("%x",(unsigned char)node_array[i].owen_certificate.rsa.
AllCertificateElementThatIsSigned[j]);
printf("\n The certificate length in char is=%d\n",j);
printf("\nsign_certificate_rsa=\n");
cotnum(node_array[i].owen_certificate.rsa.sign_certificate_rsa,stdout);
}
*/

/*RSA signature verification on the certificate*/
printf("\n-------RSA signature verification on the certificate------\n");
start=clock();
for (j=0; j<NR; j++)
for (i=0; i<NRNODE; i++)
rsa_sign_verification(
&node_array[i].ca_certificate.rsa.SubjectPublicKeyInfo.
public_key_rsa.n,
&node_array[i].ca_certificate.rsa.SubjectPublicKeyInfo.
public_key_rsa.e,
&node_array[i].owen_certificate.rsa.sign_certificate_rsa,
&node_array[i].owen_certificate.rsa.hash_certificate,
&node_array[i].val);

finish=clock();
duration=(double)(finish-start)/CLOCKS_PER_SEC;
printf("\n");
printf("Number of interaction j=%d\n",j);
printf("Number of interaction i=%d\n",i);
printf("Elapsed time is in ms:%f\n",duration*1000);
printf("\n");

mip->IOBASE=16;
/* for (i=0; i<NRNODE; i++)
{
printf("\nFor node nr: %d\n",i);
```

```
printf("\nRSA parameters for Node=%d\n",i);
printf("private key p= \n");
cotnum(node_array[i].owen_private_key.rsa.p,stdout);
printf("private key q= \n");
cotnum(node_array[i].owen_private_key.rsa.q,stdout);
printf("private key d= \n");
cotnum(node_array[i].owen_private_key.rsa.d,stdout);
printf("seed= %d\n",node_array[i].owen_private_key.rsa.seed);
printf("public key (n)=\n");
cotnum(node_array[i].owen_certificate.rsa.SubjectPublicKeyInfo.
public_key_rsa.n,stdout);
printf("new public key (e)=\n");
cotnum(node_array[i].owen_certificate.rsa.SubjectPublicKeyInfo.
public_key_rsa.e,stdout);
printf("owenAllCertificateElementThatIsSigned_rsa=\n");
for (j=0; node_array[i].owen_certificate.rsa.
AllCertificateElementThatIsSigned[j]!='\0';j++)
printf("%x",(unsigned char)node_array[i].owen_certificate.rsa.
AllCertificateElementThatIsSigned[j]);
printf("\n The certificate length in char is=%d\n",j);
printf("\nsign_certificate_rsa=\n");
cotnum(node_array[i].owen_certificate.rsa.sign_certificate_rsa,stdout);
printf("validation = %d\n",node_array[i].val);
if (node_array[i].val==1)
printf("The RSA Signature is verified\n");
else printf("The RSA Signature is NOT verified\n");
}
*/
/*RSA signature generation on the first master hash chain key*/
printf("\n-RSA generate signature on the first master hash chain key-\n");

start=clock();
for (j=0; j<NR; j++)
for (i=0; i<NRNODE; i++)
rsa_signature(
&node_array[i].owen_certificate.rsa.SubjectPublicKeyInfo.
public_key_rsa.n,
&node_array[i].owen_private_key.rsa.p,
&node_array[i].owen_private_key.rsa.q,
&node_array[i].owen_certificate.rsa.SubjectPublicKeyInfo.
public_key_rsa.e,
&node_array[i].owen_private_key.rsa.d,
&node_array[i].sign_master_hash_chain.sign_rsa.s,
&node_array[i].hash_to_id_last_master_hash_time0_lifetime_big);
finish=clock();
duration=(double)(finish-start)/CLOCKS_PER_SEC;
```

```
printf("\n");
printf("Number of interaction j=%d\n",j);
printf("Number of interaction i=%d\n",i);
printf("Elapsed time is in ms:%f\n",duration*1000);
printf("\n");

/* for (i=0; i<NRNODE; i++)
{
printf("\nFor node nr: %d\n",i);
printf("\n The RSA signature on the master hash chain=\n");
cotnum(node_array[i].sign_master_hash_chain.sign_rsa.s,stdout);
}
*/
mip->IOBASE=16;
/* for (i=0; i<NRNODE; i++)
{
printf("\nFor node nr: %d\n",i);
printf("\nRSA parameters for Node=%d\n",i);
printf("private key p= \n");
cotnum(node_array[i].owen_private_key.rsa.p,stdout);
printf("private key q= \n");
cotnum(node_array[i].owen_private_key.rsa.q,stdout);
printf("private key d= \n");
cotnum(node_array[i].owen_private_key.rsa.d,stdout);
printf("seed= %d\n",node_array[i].owen_private_key.rsa.seed);
printf("public key (n)=\n");
cotnum(node_array[i].owen_certificate.rsa.SubjectPublicKeyInfo.
public_key_rsa.n,stdout);
printf("new public key (e)=\n");
cotnum(node_array[i].owen_certificate.rsa.SubjectPublicKeyInfo.
public_key_rsa.e,stdout);
printf("The hash_to_id_last_master_hash_time0_lifetime=\n");
cotnum(node_array[i].hash_to_id_last_master_hash_time0_lifetime_big,
stdout);
printf("\nsign on the first master hash chain key=\n");
cotnum(node_array[i].sign_master_hash_chain.sign_rsa.s,stdout);
}
*/

/*RSA signature verification on the first master hash chain value
include node id and time start/stop*/
printf("\n-----RSA signature on first master hash chain key etc----\n");
start=clock();
for (j=0; j<NR; j++)
for (i=0; i<NRNODE; i++)
rsa_sign_verification(
```

```
&node_array[i].owen_certificate.rsa.SubjectPublicKeyInfo.
public_key_rsa.n,
&node_array[i].owen_certificate.rsa.SubjectPublicKeyInfo.
public_key_rsa.e,
&node_array[i].sign_master_hash_chain.sign_rsa.s,
&node_array[i].hash_to_id_last_master_hash_time0_lifetime_big,
&node_array[i].val);
finish=clock();
duration=(double)(finish-start)/CLOCKS_PER_SEC;
printf("\n");
printf("Number of interaction j=%d\n",j);
printf("Number of interaction i=%d\n",i);
printf("Elapsed time is in ms:%f\n",duration*1000);
printf("\n");

mip->IOBASE=16;
/* for (i=0; i<NRNODE; i++)
{
printf("\nFor node nr: %d\n",i);
printf("\nRSA parameters for Node=%d\n",i);
printf("private key p= \n");
cotnum(node_array[i].owen_private_key.rsa.p,stdout);
printf("private key q= \n");
cotnum(node_array[i].owen_private_key.rsa.q,stdout);
printf("private key d= \n");
cotnum(node_array[i].owen_private_key.rsa.d,stdout);
printf("seed= %d\n",node_array[i].owen_private_key.rsa.seed);
printf("public key (n)=\n");
cotnum(node_array[i].owen_certificate.rsa.SubjectPublicKeyInfo.
public_key_rsa.n,stdout);
printf("new public key (e)=\n");
cotnum(node_array[i].owen_certificate.rsa.SubjectPublicKeyInfo.
public_key_rsa.e,stdout);
printf("signature to master hash chain=\n");
cotnum(node_array[i].sign_master_hash_chain.sign_rsa.s,stdout);
printf("the hash to the first master hash chain key=\n");
cotnum(node_array[i].hash_to_id_last_master_hash_time0_lifetime_big,
stdout);
printf("validation = %d\n",node_array[i].val);
if (node_array[i].val==1)
printf("The RSA Signature is verified\n");
else printf("The RSA Signature is NOT verified\n");
}
*/
/*make break on output*/
printf("\n---- You have to tuch a key to se the rest and push return----\n");
```

```
c=getchar();


/*HMAC with SHA-1 of input data and hash key */
printf("---------------Generate HMAC------------\n");
lk_hmac_sha_1=20; /*The key size to k_hmac_sha_1 in byte*/
ld_hmac_sha_1=20; /*the length of data input in byte*/
t_hmac_sha_1=20; /*The size of output from HMAC in byte*/


start=clock();
for (j=0; j<MAX_length_MASTER_HASH_CHAIN; j++)
{
for (i=0; i<NRNODE;i++)
{
hmac_sha_1( node_array[i].Master_hash_chain[1],
&lk_hmac_sha_1,node_array[i].Traffic_hash_chain[1],
&ld_hmac_sha_1, out_hmac_sha_1, &t_hmac_sha_1);
}
}
finish=clock();
duration=(double)(finish-start)/CLOCKS_PER_SEC;
printf("\n");
printf("Number of interaction=%d\n",j);
printf("Number of node=%d\n",i);
printf("\n");
printf("Elapsed time is in ms :%f\n",duration*1000);
printf("\n");

/*-----------NODE GENERATION AND CERTIFICATE-----------------------*/
    /*ECDSA generate public and private key*/
printf("\n-Generation of ecdsa public/private key and certificate-\n");

start=clock();
for (j=0; j<NRNODE; j++)
for (i=0; i<NRNODE; i++)
ecdsa_gen( &node_array[i].owen_certificate.curve_ecdsa.a,
&node_array[i].owen_certificate.curve_ecdsa.b,
&node_array[i].owen_certificate.curve_ecdsa.p,
&node_array[i].owen_certificate.curve_ecdsa.q,
&node_array[i].owen_certificate.curve_ecdsa.x,
&node_array[i].owen_certificate.curve_ecdsa.y,
&node_array[i].owen_private_key.ecdsa.d,
&node_array[i].owen_private_key.ecdsa.seed,
&node_array[i].owen_certificate.ecdsa.SubjectPublicKeyInfo.
public_key_ecdsa.x,
```

```
&node_array[i].owen_certificate.ecdsa.SubjectPublicKeyInfo.
public_key_ecdsa.ep);
finish=clock();
duration=(double)(finish-start)/CLOCKS_PER_SEC;
printf("\n");
printf("Number of interaction j=%d\n",j);
printf("Number of interaction i=%d\n",i);
printf("Elapsed time is in ms:%f\n",duration*1000);
printf("\n");

for (i=0; i<NRNODE; i++)
{
time(&ltime); /*The UTC time that will be used in the certificate*/
node_array[i].owen_certificate.ecdsa.CertificateSerialNumber=30000+i;
node_array[i].owen_certificate.ecdsa.Time.generalizedTime=ltime;
node_array[i].owen_certificate.ecdsa.Time.utcTime=ltime;
node_array[i].owen_certificate.ecdsa.Validity.notBefore=ltime;
node_array[i].owen_certificate.ecdsa.Validity.notAfter=ltime+100000000;
node_array[i].owen_certificate.ecdsa.AlgorithmIdentifier.
ALGORITHM_id=3;
strcpy(node_array[i].owen_certificate.ecdsa.AlgorithmIdentifier.
ALGORITHM_Type,"ECDSA");
strcpy(node_array[i].owen_certificate.ecdsa.
SupportedAlgorithms,"ECDSA,SHA1");
node_array[i].owen_certificate.ecdsa.version=0;
strcpy(node_array[i].owen_certificate.ecdsa.issuer,"CA");
node_array[i].owen_certificate.ecdsa.subject=node_array[i].
owen_certificate.ecdsa.CertificateSerialNumber;
strcpy(node_array[i].owen_certificate.ecdsa.SubjectPublicKeyInfo.
algorithm,"ECDSA");
}
/*Signature on ECDSA certificate*/
printf("\n----Signature on ECDSA certificate----\n");

start=clock();
for (j=0; j<NR; j++)
{
for (i=0; i<NRNODE; i++)
{
hash_of_ecdsa_certificate(
node_array[i].owen_certificate.ecdsa.version,
node_array[i].owen_certificate.ecdsa.CertificateSerialNumber,
node_array[i].owen_certificate.ecdsa.AlgorithmIdentifier.
ALGORITHM_id,
node_array[i].owen_certificate.ecdsa.AlgorithmIdentifier.
ALGORITHM_Type,
```

```
node_array[i].owen_certificate.ecdsa.issuer,
node_array[i].owen_certificate.ecdsa.Validity.notBefore,
node_array[i].owen_certificate.ecdsa.Validity.notAfter,
node_array[i].owen_certificate.ecdsa.subject,
node_array[i].owen_certificate.ecdsa.SubjectPublicKeyInfo.
algorithm,
node_array[i].owen_certificate.ecdsa.SubjectPublicKeyInfo.
public_key_ecdsa.x,
node_array[i].owen_certificate.ecdsa.SubjectPublicKeyInfo.
public_key_ecdsa.ep,
node_array[i].owen_certificate.ecdsa.hash_certificate,
node_array[i].owen_certificate.ecdsa.
AllCertificateElementThatIsSigned);
/*Node "0" is the CA*/

ecdsa_sign(
&node_array[0].owen_certificate.curve_ecdsa.a,
&node_array[0].owen_certificate.curve_ecdsa.b,
&node_array[0].owen_certificate.curve_ecdsa.p,
&node_array[0].owen_certificate.curve_ecdsa.q,
&node_array[0].owen_certificate.curve_ecdsa.x,
&node_array[0].owen_certificate.curve_ecdsa.y,
&node_array[0].owen_private_key.ecdsa.d,
&node_array[0].owen_private_key.ecdsa.seed,
&node_array[0].owen_certificate.ecdsa.
SubjectPublicKeyInfo.public_key_ecdsa.x,
&node_array[i].owen_certificate.ecdsa.sign_certificate_ecdsa.r,
&node_array[i].owen_certificate.ecdsa.sign_certificate_ecdsa.s,
&node_array[i].owen_certificate.ecdsa.hash_certificate);
}
}
finish=clock();
duration=(double)(finish-start)/CLOCKS_PER_SEC;
printf("\n");
printf("Number of interaction j=%d\n",j);
printf("Number of interaction i=%d\n",i);
printf("Elapsed time is in ms:%f\n",duration*1000);
printf("\n");

for (i=0; i<NRNODE; i++)
for (j=0; node_array[i].owen_certificate.ecdsa.
AllCertificateElementThatIsSigned[j]!='\0';j++);
printf("\n The lengt to ECDSA certificate is=%d\n",j);

/* printf("\n CA certificate signature_s=\n");
cotnum(node_array[0].owen_certificate.ecdsa.sign_certificate_ecdsa.s,
```

```
stdout);
*/
for (i=0; i<NRNODE; i++)
{
node_array[i].ca_certificate.ecdsa=node_array[0].owen_certificate.
ecdsa;
/*printf("\n Node n CA certificate signature_s=%d\n",i);
cotnum(node_array[i].ca_certificate.ecdsa.sign_certificate_ecdsa.s,
stdout);
*/
}

mip->IOBASE=16;
/* for (i=0; i<NRNODE; i++)
{
printf("\nECC parameters for Node=%d\n",i);
printf("\n");
printf("a= \n");
cotnum(node_array[i].owen_certificate.curve_ecdsa.a,stdout);
printf("b= \n");
cotnum(node_array[i].owen_certificate.curve_ecdsa.b,stdout);
printf("p= \n");
cotnum(node_array[i].owen_certificate.curve_ecdsa.p,stdout);
printf("q= \n");
cotnum(node_array[i].owen_certificate.curve_ecdsa.q,stdout);
printf("Public key from the courv (x)= \n");
cotnum(node_array[i].owen_certificate.curve_ecdsa.x,stdout);
printf("y= \n");
cotnum(node_array[i].owen_certificate.curve_ecdsa.y,stdout);
printf("new private key d= \n");
cotnum(node_array[i].owen_private_key.ecdsa.d,stdout);
printf("seed= %d\n",node_array[i].owen_private_key.ecdsa.seed);
printf("new public key (x)=\n");
cotnum(node_array[i].owen_certificate.ecdsa.SubjectPublicKeyInfo.
public_key_ecdsa.x,stdout);
printf("ep= %d\n",node_array[i].owen_certificate.ecdsa.
SubjectPublicKeyInfo.public_key_ecdsa.ep);
printf("owenAllCertificateElementThatIsSigned_ecdsa=\n");
for (j=0; node_array[i].owen_certificate.ecdsa.
AllCertificateElementThatIsSigned[j]!='\0';j++)
printf("%x",(unsigned char)node_array[i].owen_certificate.ecdsa.
AllCertificateElementThatIsSigned[j]);
printf("\n The certificate length in char is=%d\n",j);
printf("\nsign_certificate_ecdsa.r=\n");
cotnum(node_array[i].owen_certificate.ecdsa.sign_certificate_ecdsa.r,
stdout);
```

```
printf("sign_certificate_ecdsa.s=\n");
cotnum(node_array[i].owen_certificate.ecdsa.sign_certificate_ecdsa.s,
stdout);
}
*/

/*ECDSA signature verification on the certificate*/
printf("\n------ECDSA signature verification on certificate----\n");

start=clock();
for (j=0; j<NR; j++)
for (i=0; i<NRNODE; i++)
ecdsa_ver(
&node_array[i].owen_certificate.curve_ecdsa.a,
&node_array[i].owen_certificate.curve_ecdsa.b,
&node_array[i].owen_certificate.curve_ecdsa.p,
&node_array[i].owen_certificate.curve_ecdsa.q,
&node_array[i].owen_certificate.curve_ecdsa.x,
&node_array[i].owen_certificate.curve_ecdsa.y,
&node_array[i].owen_private_key.ecdsa.d,
&node_array[i].owen_private_key.ecdsa.seed,
&node_array[i].owen_certificate.ecdsa.
SubjectPublicKeyInfo.public_key_ecdsa.x,
&node_array[i].owen_certificate.ecdsa.
SubjectPublicKeyInfo.public_key_ecdsa.ep,
&node_array[i].owen_certificate.ecdsa.sign_certificate_ecdsa.r,
&node_array[i].owen_certificate.ecdsa.sign_certificate_ecdsa.s,
&node_array[i].owen_certificate.ecdsa.hash_certificate,
&node_array[i].val);

finish=clock();
duration=(double)(finish-start)/CLOCKS_PER_SEC;
printf("\n");
printf("Number of interaction j=%d\n",j);
printf("Number of interaction i=%d\n",i);
printf("Elapsed time is in ms:%f\n",duration*1000);
printf("\n");


mip->IOBASE=16;
/* for (i=0; i<NRNODE; i++)
{
printf("a= \n");
cotnum(node_array[i].owen_certificate.curve_ecdsa.a,stdout);
printf("b= \n");
cotnum(node_array[i].owen_certificate.curve_ecdsa.b,stdout);
```

```
printf("p= \n");
cotnum(node_array[i].owen_certificate.curve_ecdsa.p,stdout);
printf("q= \n");
cotnum(node_array[i].owen_certificate.curve_ecdsa.q,stdout);
printf("Public key from the courv (x)= \n");
cotnum(node_array[i].owen_certificate.curve_ecdsa.x,stdout);
printf("y= \n");
cotnum(node_array[i].owen_certificate.curve_ecdsa.y,stdout);
printf("new private key d= \n");
cotnum(node_array[i].owen_private_key.ecdsa.d,stdout);
printf("seed= %d\n",node_array[i].owen_private_key.ecdsa.seed);
printf("new public key (x)= \n");
cotnum(node_array[i].owen_certificate.ecdsa.SubjectPublicKeyInfo.
public_key_ecdsa.x,stdout);
printf("node_array[i].sign_master_hash_chain_r= \n");
cotnum(node_array[i].sign_master_hash_chain.sign_ecdsa.r,stdout);
printf("node_array[i].sign_master_hash_chain_s= \n");
cotnum(node_array[i].sign_master_hash_chain.sign_ecdsa.s,stdout);
printf("node_array[i].hash_id_last_master_hash_time0_lifetime_big \n");
cotnum(node_array[i].hash_to_id_last_master_hash_time0_lifetime_big,
stdout);
printf("ep= %d\n",node_array[i].owen_certificate.ecdsa.
SubjectPublicKeyInfo.public_key_ecdsa.ep);
printf("val_ecdsa=%df\n",node_array[i].val);
if (node_array[i].val==1)
printf("Signature is verified\n");
else
printf("Signature is NOT verified\n");
}
*/



/*NODE ECDSA signature generation to the master hash chain*/
printf("\nECDSA generate signature on the first master hash chain key\n");

start=clock();
for (j=0; j<NR; j++)
for (i=0; i<NRNODE; i++)
ecdsa_sign(
&node_array[i].owen_certificate.curve_ecdsa.a,
&node_array[i].owen_certificate.curve_ecdsa.b,
&node_array[i].owen_certificate.curve_ecdsa.p,
&node_array[i].owen_certificate.curve_ecdsa.q,
&node_array[i].owen_certificate.curve_ecdsa.x,
&node_array[i].owen_certificate.curve_ecdsa.y,
```

```
&node_array[i].owen_private_key.ecdsa.d,
&node_array[i].owen_private_key.ecdsa.seed,
&node_array[i].owen_certificate.ecdsa.
SubjectPublicKeyInfo.public_key_ecdsa.x,
&node_array[i].sign_master_hash_chain.sign_ecdsa.r,
&node_array[i].sign_master_hash_chain.sign_ecdsa.s,
&node_array[i].hash_to_id_last_master_hash_time0_lifetime_big);
finish=clock();
duration=(double)(finish-start)/CLOCKS_PER_SEC;
printf("\n");
printf("Number of interaction j=%d\n",j);
printf("Number of interaction i=%d\n",i);
printf("Elapsed time is in ms:%f\n",duration*1000);
printf("\n");


mip->IOBASE=16;
/* for (i=0; i<NRNODE; i++)
{
printf("a= \n");
cotnum(node_array[i].owen_certificate.curve_ecdsa.a,stdout);
printf("b= \n");
cotnum(node_array[i].owen_certificate.curve_ecdsa.b,stdout);
printf("p= \n");
cotnum(node_array[i].owen_certificate.curve_ecdsa.p,stdout);
printf("q= \n");
cotnum(node_array[i].owen_certificate.curve_ecdsa.q,stdout);
printf("Public key from the courv (x)= \n");
cotnum(node_array[i].owen_certificate.curve_ecdsa.x,stdout);
printf("y= \n");
cotnum(node_array[i].owen_certificate.curve_ecdsa.y,stdout);
printf("new private key d= \n");
cotnum(node_array[i].owen_private_key.ecdsa.d,stdout);
printf("seed= %d\n",node_array[i].owen_private_key.ecdsa.seed);
printf("new public key (x)=\n");
cotnum(node_array[i].owen_certificate.ecdsa.SubjectPublicKeyInfo.
public_key_ecdsa.x,stdout);
printf("sign_master_hash_chain_r= \n");
cotnum(node_array[i].sign_master_hash_chain.sign_ecdsa.r,stdout);
printf("sign_master_hash_chain_s= \n");
cotnum(node_array[i].sign_master_hash_chain.sign_ecdsa.s,stdout);
printf("node_array[i].hash_id_last_master_hash_time0_lifetime_big \n");
cotnum(node_array[i].hash_to_id_last_master_hash_time0_lifetime_big,
stdout);
}
*/
```

```
/*ECDSA signature verification on the first master hash chain key*/
printf("\nECDSA signature verification to first master hash chain key\n");

start=clock();
for (j=0; j<NR; j++)
for (i=0; i<NRNODE; i++)
ecdsa_ver(
&node_array[i].owen_certificate.curve_ecdsa.a,
&node_array[i].owen_certificate.curve_ecdsa.b,
&node_array[i].owen_certificate.curve_ecdsa.p,
&node_array[i].owen_certificate.curve_ecdsa.q,
&node_array[i].owen_certificate.curve_ecdsa.x,
&node_array[i].owen_certificate.curve_ecdsa.y,
&node_array[i].owen_private_key.ecdsa.d,
&node_array[i].owen_private_key.ecdsa.seed,
&node_array[i].owen_certificate.ecdsa.
SubjectPublicKeyInfo.public_key_ecdsa.x,
&node_array[i].owen_certificate.ecdsa.
SubjectPublicKeyInfo.public_key_ecdsa.ep,
&node_array[i].sign_master_hash_chain.sign_ecdsa.r,
&node_array[i].sign_master_hash_chain.sign_ecdsa.s,
&node_array[i].hash_to_id_last_master_hash_time0_lifetime_big,
&node_array[i].val);
finish=clock();
duration=(double)(finish-start)/CLOCKS_PER_SEC;
printf("\n");
printf("Number of interaction j=%d\n",j);
printf("Number of interaction i=%d\n",i);
printf("Elapsed time is in ms:%f\n",duration*1000);
printf("\n");


mip->IOBASE=16;
/* for (i=0; i<NRNODE; i++)
{
printf("a= \n");
cotnum(node_array[i].owen_certificate.curve_ecdsa.a,stdout);
printf("b= \n");
cotnum(node_array[i].owen_certificate.curve_ecdsa.b,stdout);
printf("p= \n");
cotnum(node_array[i].owen_certificate.curve_ecdsa.p,stdout);
printf("q= \n");
cotnum(node_array[i].owen_certificate.curve_ecdsa.q,stdout);
printf("Public key from the courv (x)= \n");
cotnum(node_array[i].owen_certificate.curve_ecdsa.x,stdout);
printf("y= \n");
```

```
cotnum(node_array[i].owen_certificate.curve_ecdsa.y,stdout);
printf("new private key d= \n");
cotnum(node_array[i].owen_private_key.ecdsa.d,stdout);
printf("seed= %d\n",node_array[i].owen_private_key.ecdsa.seed);
printf("new public key (x)= \n");
cotnum(node_array[i].owen_certificate.ecdsa.SubjectPublicKeyInfo.
public_key_ecdsa.x,stdout);
printf("node_array[i].sign_master_hash_chain_r= \n");
cotnum(node_array[i].sign_master_hash_chain.sign_ecdsa.r,stdout);
printf("node_array[i].sign_master_hash_chain_s= \n");
cotnum(node_array[i].sign_master_hash_chain.sign_ecdsa.s,stdout);
printf("node_array[i].hash_id_last_master_hash_time0_lifetime_big \n");
cotnum(node_array[i].hash_to_id_last_master_hash_time0_lifetime_big,
stdout);
printf("ep= %d\n",node_array[i].owen_certificate.ecdsa.
SubjectPublicKeyInfo.public_key_ecdsa.ep);
printf("val_ecdsa=%df\n",node_array[i].val);
if (node_array[i].val==1)
printf("Signature is verified\n");
else
printf("Signature is NOT verified\n");
}
*/
/*Test og message 1 when using DSA*/
printf("\n-------Test of message 1 when using DSA-------\n");

start=clock();
for (j=0; j<NR3; j++)
{
for (i=0; i<NRNODE; i++)
{
time(&ltime);
hash_time=ltime-node_array[i].master_hash_key_time0;
message1_ecdsa_dsa(
node_array[i].owen_certificate.dsa.
AllCertificateElementThatIsSigned,
node_array[i].owen_certificate.dsa.sign_certificate_dsa.r,
node_array[i].owen_certificate.dsa.sign_certificate_dsa.s,
node_array[i].id_last_master_hash_time0_lifetime_in_array,
node_array[i].sign_master_hash_chain.sign_dsa.r,
node_array[i].sign_master_hash_chain.sign_dsa.s,
node_array[i].transmit.challenges.
challengeStringFromInitiator,
node_array[i].transmit.challenges.
challengeFromInitiatorInteger,
node_array[i].Master_hash_chain[hash_time],
```

```
node_array[i].Traffic_hash_chain[node_array[i].index_kTraffic],
node_array[i].Master_hash_chain[++hash_time],
node_array[i].transmit.mac,
node_array[i].transmit.message_array);
}
}
finish=clock();
duration=(double)(finish-start)/CLOCKS_PER_SEC;
printf("\n");
printf("Number of interaction j=%d\n",j);
printf("Number of interaction i=%d\n",i);
printf("Elapsed time is in ms:%f\n",duration*1000);
printf("\n");

for (i=0; i<NRNODE; i++)
{
printf("\n Message to be transmitted from node nr:%d\n",i);
printf("\n Message that is transmitted:\n");
for (j=0;node_array[i].transmit.message_array[j]!='\0';j++)
printf("%x",(unsigned char)node_array[i].
transmit.message_array[j]);
printf("\n The message length in char is=%d\n",j);
}

/*Test og message 2 when using DSA*/
printf("\n-----------Test of message 2 when using DSA----------\n");
start=clock();
for (j=0; j<NR3; j++)
{
for (i=0; i<NRNODE; i++)
{
time(&ltime);
hash_time=ltime-node_array[i].master_hash_key_time0;
message2_ecdsa_dsa(
node_array[i].owen_certificate.dsa.
AllCertificateElementThatIsSigned,
node_array[i].owen_certificate.dsa.sign_certificate_dsa.r,
node_array[i].owen_certificate.dsa.sign_certificate_dsa.s,
node_array[i].id_last_master_hash_time0_lifetime_in_array,
node_array[i].sign_master_hash_chain.sign_dsa.r,
node_array[i].sign_master_hash_chain.sign_dsa.s,
node_array[i].transmit.challenges.
challengeStringFromResponder,
node_array[i].transmit.challenges.
challengeFromResponderInteger,
node_array[i].Master_hash_chain[hash_time],
```

```
node_array[i].Traffic_hash_chain[node_array[i].index_kTraffic],
node_array[i].Master_hash_chain[++hash_time],
node_array[i].transmit.mac,
node_array[i].transmit.message_array,
node_array[i].transmit.challenges.challengeStringFromInitiator,
node_array[i].transmit.challenges.
hashOfchallengeStringFromResponder);
}
}
finish=clock();
duration=(double)(finish-start)/CLOCKS_PER_SEC;
printf("\n");
printf("Number of interaction j=%d\n",j);
printf("Number of interaction i=%d\n",i);
printf("Elapsed time is in ms:%f\n",duration*1000);
printf("\n");

for (i=0; i<NRNODE; i++)
{
printf("\n Message to be transmitted from node nr:%d\n",i);
printf("\n Message that is transmitted:\n");
for (j=0;node_array[i].transmit.message_array[j]!='\0';j++)
printf("%x",(unsigned char)node_array[i].transmit.
message_array[j]);
printf("\n The message length in char is=%d\n",j);
}


/*Test og message 1 when using RSA*/
printf("\n--------------Test of message 1 when using RSA-------------\n");

start=clock();
for (j=0; j<NR3; j++)
{
for (i=0; i<NRNODE; i++)
{
time(&ltime);
hash_time=ltime-node_array[i].master_hash_key_time0;
message1_rsa(
node_array[i].owen_certificate.rsa.
AllCertificateElementThatIsSigned,
node_array[i].owen_certificate.rsa.sign_certificate_rsa,
node_array[i].id_last_master_hash_time0_lifetime_in_array,
node_array[i].sign_master_hash_chain.sign_rsa.s,
node_array[i].transmit.challenges.
challengeStringFromInitiator,
```

```
node_array[i].transmit.challenges.
challengeFromInitiatorInteger,
node_array[i].Master_hash_chain[hash_time],
node_array[i].Traffic_hash_chain[node_array[i].index_kTraffic],
node_array[i].Master_hash_chain[++hash_time],
node_array[i].transmit.mac,
node_array[i].transmit.message_array);
}
}
finish=clock();
duration=(double)(finish-start)/CLOCKS_PER_SEC;
printf("\n");
printf("Number of interaction j=%d\n",j);
printf("Number of interaction i=%d\n",i);
printf("Elapsed time is in ms:%f\n",duration*1000);
printf("\n");

for (i=0; i<NRNODE; i++)
{
printf("\n Message to be transmitted from node nr:%d\n",i);
printf("\n Message that is transmitted:\n");
for (j=0;node_array[i].transmit.message_array[j]!='\0';j++)
printf("%x",(unsigned char)node_array[i].transmit.
message_array[j]);
printf("\n The message length in char is=%d\n",j);
}

/*Test og message 2 when using RSA*/
printf("\n--------------Test of message 2 when using RSA-------------\n");
start=clock();
for (j=0; j<NR3; j++)
{
for (i=0; i<NRNODE; i++)
{
time(&ltime);
hash_time=ltime-node_array[i].master_hash_key_time0;
message2_rsa(
node_array[i].owen_certificate.rsa.
AllCertificateElementThatIsSigned,
node_array[i].owen_certificate.rsa.sign_certificate_rsa,
node_array[i].id_last_master_hash_time0_lifetime_in_array,
node_array[i].sign_master_hash_chain.sign_rsa.s,
node_array[i].transmit.challenges.
challengeStringFromResponder,
node_array[i].transmit.challenges.
challengeFromResponderInteger,
```

```
node_array[i].Master_hash_chain[hash_time],
node_array[i].Traffic_hash_chain[node_array[i].index_kTraffic],
node_array[i].Master_hash_chain[++hash_time],
node_array[i].transmit.mac,
node_array[i].transmit.message_array,
node_array[i].transmit.challenges.challengeStringFromInitiator,
node_array[i].transmit.challenges.
hashOfchallengeStringFromResponder);
}
}
finish=clock();
duration=(double)(finish-start)/CLOCKS_PER_SEC;
printf("\n");
printf("Number of interaction j=%d\n",j);
printf("Number of interaction i=%d\n",i);
printf("elapsed time is in ms:%f\n",duration*1000);
printf("\n");

for (i=0; i<NRNODE; i++)
{
printf("\n Message to be transmitted from node nr:%d\n",i);
printf("\n Message that is transmitted:\n");
for (j=0;node_array[i].transmit.message_array[j]!='\0';j++)
printf("%x",(unsigned char)node_array[i].transmit.
message_array[j]);
printf("\n The message length in char is=%d\n",j);
}



/*Test og message 1 when using ECDSA*/
printf("\n----Test of message 1 when using ECDSA------\n");

start=clock();
for (j=0; j<NR3; j++)
{
for (i=0; i<NRNODE; i++)
{
time(&ltime);
hash_time=ltime-node_array[i].master_hash_key_time0;
message1_ecdsa_dsa(
node_array[i].owen_certificate.ecdsa.
AllCertificateElementThatIsSigned,
node_array[i].owen_certificate.ecdsa.sign_certificate_ecdsa.r,
node_array[i].owen_certificate.ecdsa.sign_certificate_ecdsa.s,
node_array[i].id_last_master_hash_time0_lifetime_in_array,
```

```
node_array[i].sign_master_hash_chain.sign_ecdsa.r,
node_array[i].sign_master_hash_chain.sign_ecdsa.s,
node_array[i].transmit.challenges.
challengeStringFromInitiator,
node_array[i].transmit.challenges.
challengeFromInitiatorInteger,
node_array[i].Master_hash_chain[hash_time],
node_array[i].Traffic_hash_chain[node_array[i].index_kTraffic],
node_array[i].Master_hash_chain[++hash_time],
node_array[i].transmit.mac,
node_array[i].transmit.message_array);
}
}
finish=clock();
duration=(double)(finish-start)/CLOCKS_PER_SEC;
printf("\n");
printf("Number of interaction j=%d\n",j);
printf("Number of interaction i=%d\n",i);
printf("Elapsed time is in ms:%f\n",duration*1000);
printf("\n");

for (i=0; i<NRNODE; i++)
{
printf("\n Message to be transmitted from node nr:%d\n",i);
printf("\n Message that is transmitted:\n");
for (j=0;node_array[i].transmit.message_array[j]!='\0';j++)
printf("%x",(unsigned char)node_array[i].transmit.
message_array[j]);
printf("\n The message length in char is=%d\n",j);
}

/*Test og message 2 when using ECDSA*/
printf("\n-------Test of message 2 when using ECDSA-------\n");

start=clock();
for (j=0; j<NR3; j++)
{
for (i=0; i<NRNODE; i++)
{
time(&ltime);
hash_time=ltime-node_array[i].master_hash_key_time0;
message2_ecdsa_dsa(
node_array[i].owen_certificate.ecdsa.
AllCertificateElementThatIsSigned,
node_array[i].owen_certificate.ecdsa.sign_certificate_ecdsa.r,
node_array[i].owen_certificate.ecdsa.sign_certificate_ecdsa.s,
```

```
node_array[i].id_last_master_hash_time0_lifetime_in_array,
node_array[i].sign_master_hash_chain.sign_ecdsa.r,
node_array[i].sign_master_hash_chain.sign_ecdsa.s,
node_array[i].transmit.challenges.
challengeStringFromResponder,
node_array[i].transmit.challenges.
challengeFromResponderInteger,
node_array[i].Master_hash_chain[hash_time],
node_array[i].Traffic_hash_chain[node_array[i].index_kTraffic],
node_array[i].Master_hash_chain[++hash_time],
node_array[i].transmit.mac,
node_array[i].transmit.message_array,
node_array[i].transmit.challenges.challengeStringFromInitiator,
node_array[i].transmit.challenges.
hashOfchallengeStringFromResponder);
}
}
finish=clock();
duration=(double)(finish-start)/CLOCKS_PER_SEC;
printf("\n");
printf("Number of interaction j=%d\n",j);
printf("Number of interaction i=%d\n",i);
printf("Elapsed time is in ms:%f\n",duration*1000);
printf("\n");

for (i=0; i<NRNODE; i++)
{
printf("\n Message to be transmitted from node nr:%d\n",i);
printf("\n Message that is transmitted:\n");
for (j=0;node_array[i].transmit.message_array[j]!='\0';j++)
printf("%x",(unsigned char)node_array[i].transmit.
message_array[j]);
printf("\n The message length in char is=%d\n",j);
}

/*Test og message 3*/
printf("\n--------------Test of message 3-------------\n");
/* for (i=0; i<NRNODE; i++)
{
printf("\n challengeStringFromResponder:\n");
for (j=0;node_array[i].transmit.challenges.
challengeStringFromResponder[j]!='\0';j++)
printf("%x",(unsigned char)node_array[i].transmit.
challenges.challengeStringFromResponder[j]);

printf("\n challengeStringFromInitiator:\n");
```

```
for (j=0;node_array[i].transmit.challenges.
challengeStringFromInitiator[j]!='\0';j++)
printf("%x",(unsigned char)node_array[i].transmit.
challenges.challengeStringFromInitiator[j]);
}
*/
start=clock();
for (j=0; j<NR3; j++)
{
for (i=0; i<NRNODE; i++)
{
time(&ltime);
hash_time=ltime-node_array[i].master_hash_key_time0;
message3(
node_array[i].owen_certificate.ecdsa.CertificateSerialNumber,
node_array[i].owen_certificate.ecdsa.CertificateSerialNumber,
node_array[i].transmit.challenges.
hashOfchallengeStringFromResponder,
node_array[i].transmit.challenges.
challengeStringFromInitiator,
node_array[i].transmit.challenges.
InitiatorsHashOfrespondsTochallengeFromResponder,
node_array[i].
Traffic_hash_chain[node_array[i].index_kTraffic++],
node_array[i].transmit.mac,
node_array[i].transmit.message_array);
}
}
finish=clock();
duration=(double)(finish-start)/CLOCKS_PER_SEC;
printf("\n");
printf("Number of interaction j=%d\n",j);
printf("Number of interaction i=%d\n",i);
printf("Elapsed time is in ms:%f\n",duration*1000);
printf("\n");

for (i=0; i<NRNODE; i++)
{
printf("\n Message to be transmitted from node nr:%d\n",i);
printf("\n Message that is transmitted:\n");
for (j=0;node_array[i].transmit.message_array[j]!='\0';j++)
printf("%x",(unsigned char)node_array[i].transmit.
message_array[j]);
printf("\n The message length in char is=%d\n",j);
}
```

```
/*Test og message 4 and 8*/
printf("\n--------------Test of message 4 and 8-------------\n");

start=clock();
for (j=0; j<NR4; j++)
{
for (i=0; i<NRNODE; i++)
{
message4And8(
node_array[i].owen_certificate.ecdsa.CertificateSerialNumber,
node_array[i].
Traffic_hash_chain[node_array[i].index_kTraffic++],
node_array[i].transmit.message_array);
}
}
finish=clock();
duration=(double)(finish-start)/CLOCKS_PER_SEC;
printf("\n");
printf("Number of interaction j=%d\n",j);
printf("Number of interaction i=%d\n",i);
printf("Elapsed time is in ms:%f\n",duration*1000);
printf("\n");

for (i=0; i<NRNODE; i++)
{
printf("\n Message nr 4 and 8 to be transmitted from node nr:%d\n",i);
printf("\n Message nr 4 and 8 that is transmitted:\n");
for (j=0;node_array[i].transmit.message_array[j]!='\0';j++)
printf("%x",(unsigned char)node_array[i].transmit.
message_array[j]);
printf("\n The message length in char is=%d\n",j);
}

/*Test og message 5, 9, 11 and 13*/
printf("\n--------------Test of message 5, 9, 11 and 13-------------\n");
start=clock();
for (j=0; j<NR3; j++)
{
for (i=0; i<NRNODE; i++)
{
time(&ltime);
hash_time=ltime-node_array[i].master_hash_key_time0;
message5And9And11And13(
node_array[i].owen_certificate.ecdsa.CertificateSerialNumber,
node_array[i].Traffic_hash_chain[node_array[i].index_kTraffic],
node_array[i].Master_hash_chain[hash_time],
```

```
node_array[i].Master_hash_chain[++hash_time],
node_array[i].transmit.mac,
node_array[i].transmit.message_array);
}
}
finish=clock();
duration=(double)(finish-start)/CLOCKS_PER_SEC;
printf("\n");
printf("Number of interaction j=%d\n",j);
printf("Number of interaction i=%d\n",i);
printf("Elapsed time is in ms:%f\n",duration*1000);
printf("\n");
for (i=0; i<NRNODE; i++)
{
printf("\n Message nr 5,9,11 and 13 from node nr:%d\n",i);
printf("\n Message nr 5, 9, 11 and 13 that is transmitted:\n");
for (j=0;node_array[i].transmit.message_array[j]!='\0';j++)
printf("%x",(unsigned char)node_array[i].transmit.
message_array[j]);
printf("\n The message length in char is=%d\n",j);
}

/*Test og message 6, 10 and 14*/
printf("\n--------------Test of message 6, 10 and 14-------------\n");
start=clock();
for (j=0; j<NR3; j++)
{
for (i=0; i<NRNODE; i++)
{
time(&ltime);
hash_time=ltime-node_array[i].master_hash_key_time0;
message6And10And14(
node_array[i].owen_certificate.ecdsa.CertificateSerialNumber,
node_array[i].
Traffic_hash_chain[node_array[i].index_kTraffic],
node_array[i].Master_hash_chain[hash_time],
node_array[i].Master_hash_chain[++hash_time],
node_array[i].transmit.mac,
node_array[i].transmit.message_array);
}
}
finish=clock();
duration=(double)(finish-start)/CLOCKS_PER_SEC;
printf("\n");
printf("Number of interaction j=%d\n",j);
printf("Number of interaction i=%d\n",i);
```

```
printf("Elapsed time is in ms:%f\n",duration*1000);
printf("\n");
for (i=0; i<NRNODE; i++)
{
printf("\n Message nr 6,10 and 14 from node nr:%d\n",i);
printf("\n Message nr 6, 10 and 14 that is transmitted:\n");
for (j=0;node_array[i].transmit.message_array[j]!='\0';j++)
printf("%x",(unsigned char)node_array[i].transmit.
message_array[j]);
printf("\n The message length in char is=%d\n",j);
}

/*Test og message 7*/
printf("\n--------------Test of message 7-------------\n");
for (i=0; i<NRNODE; i++)
strcpy(node_array[i].transmit.data_array,
"This is the hop-by-hop authentication protocol Final");
start=clock();
for (j=0; j<NR4; j++)
{
for (i=0; i<NRNODE; i++)
{
time(&ltime);
hash_time=ltime-node_array[i].master_hash_key_time0;
message7(
node_array[i].owen_certificate.ecdsa.CertificateSerialNumber,
node_array[i].transmit.data_array,
node_array[i].
Traffic_hash_chain[++node_array[i].index_kTraffic],
node_array[i].transmit.mac,
node_array[i].transmit.message_array);
}
}
finish=clock();
duration=(double)(finish-start)/CLOCKS_PER_SEC;
printf("\n");
printf("Number of interaction j=%d\n",j);
printf("Number of interaction i=%d\n",i);
printf("Elapsed time is in ms:%f\n",duration*1000);
printf("\n");

for (i=0; i<NRNODE; i++)
{
printf("\n Message nr 7 to be transmitted from node nr:%d\n",i);
printf("\n Message nr 7 that is transmitted:\n");
for (j=0;node_array[i].transmit.message_array[j]!='\0';j++)
```

```
printf("%x",(unsigned char)node_array[i].transmit.
message_array[j]);
printf("\n The message length in char is=%d\n",j);
}

/*Test og message 12*/
printf("\n--------------Test of message 12-------------\n");
for (i=0; i<NRNODE; i++)
strcpy(node_array[i].transmit.message_array,
"This is the hop-by-hop authentication protocol Final");

start=clock();
for (j=0; j<NR3; j++)
{
for (i=0; i<NRNODE; i++)
{
time(&ltime);
hash_time=ltime-node_array[i].master_hash_key_time0;
message12(
node_array[i].owen_certificate.ecdsa.CertificateSerialNumber,
node_array[i].
Traffic_hash_chain[node_array[i].index_kTraffic],
node_array[i].Master_hash_chain[++hash_time],
node_array[i].transmit.mac,
node_array[i].transmit.message_array);
}
}
finish=clock();
duration=(double)(finish-start)/CLOCKS_PER_SEC;
printf("\n");
printf("Number of interaction j=%d\n",j);
printf("Number of interaction i=%d\n",i);
printf("Elapsed time is in ms:%f\n",duration*1000);
printf("\n");
for (i=0; i<NRNODE; i++)
{
printf("\n Message nr 12 from node nr:%d\n",i);
printf("\n Message nr 12 that is transmitted:\n");
for (j=0;node_array[i].transmit.message_array[j]!='\0';j++)
printf("%x",(unsigned char)node_array[i].transmit.
message_array[j]);
printf("\n The message length in char is=%d\n",j);
}


for (i=0; i<NRNODE; i++)
```

```
{
printf("\n id_last_master_hash_time0_lifetime_in_array:\n");
for (j=0;node_array[i].
id_last_master_hash_time0_lifetime_in_array[j]!='\0';j++)
printf("%x",(unsigned char)node_array[i].
id_last_master_hash_time0_lifetime_in_array[j]);
printf("\n The length in char is=%d\n",j);
}

return 0;
}
```