Inhoud

[OTR 1](#_Toc434829950)

[OTR deel 2 2](#_Toc434829951)

[**Implementation** 2](#_Toc434829952)

[**Authentication** 3](#_Toc434829953)

[TLS Protocol 3](#_Toc434829954)

[Het diffie-hellman-sleuteluitwisselingsprotocol 4](#_Toc434829955)

[AES (Rijndael) 5](#_Toc434829956)

[**Werking** 5](#_Toc434829957)

[RSA is een asymmetrisch encryptiealgoritme 7](#_Toc434829958)

[**Werking** 7](#_Toc434829959)

[**Sleutels** 7](#_Toc434829960)

[**Versleutelen** 7](#_Toc434829961)

[**Ontcijferen** 7](#_Toc434829962)

[**Voorbeeld** 8](#_Toc434829963)

[**Ondertekenen** 8](#_Toc434829964)

[Message authentication code 9](#_Toc434829965)

[An Example of Message Authentication Code Algorithm 9](#_Toc434829966)

[Digital Signature Algorithm 10](#_Toc434829967)

[**Key generation** 10](#_Toc434829968)

[**Parameter generation** 10](#_Toc434829969)

[**Per-user keys** 10](#_Toc434829970)

[**Signing** 10](#_Toc434829971)

[**Verifying** 11](#_Toc434829972)

[**Correctness of the algorithm** 11](#_Toc434829973)

[SMP Socialist millionaire 12](#_Toc434829974)

[Off The Record Messaging protocol 12](#_Toc434829975)

# OTR

DSA:

DSA word alleen gebruikt voor authenicatie, zodat jij kan zeggen dat Bob ook echt Bob is. Word verder nergens gebruikt in het protocol dan alleen bij opstarten sessie.

Secure verbinding opzetten OTR:

1. Alice stuurt message naar Bob dat ze in OTR wilt praten, met daarin de protocol versies die ze ondersteunt.

2. Bob ondersteunt protocol en wilt met alice in OTR communiceren, hij start de Authenicated Key Exchange, door een Diffie Hellman commit te sturen, zodat Alice een key kan sturen. ( Bob stuurt zelf geen key!)

3. Alice stuurt bericht naar Bob, met daarin de key.

4. Bob stuurt zijn key naar Alice, en autheniceert zichzelf ( + channel properties ) om man in the middle te voorkomen. Hierin zit ook de eerste MAC.

5. Alice autheniceert haarzelf ( + channel properties ). Hierin zit ook de eerste MAC. Deze word gebruikt om de volgende bericht te ontcijferen.

Data verzenden naar elkaar, NA de secure verbinding opzetten, vanuit Alice gezien:

-Alice’s D-H key die is laatst acknowledged door Bob

-Als het de nieuwste D-H key is, genereer nieuwe

-Pak meest recente D-H key van bob, en genereer shared secret, sending AES en sending MAC

-Pak alle oude MAC keys gebruikt, en stuur deze mee ( Zit ook nieuwe DH key bij ), encrypted door AES.

Bob ontvangt Alice’s bericht::

-Genereert shared secret aan de hand van laatste D-H key van Bob en laatste D-H key van Alice. Hiervan word een receiving AES en receiving MAC gegenereerd,

-receiving MAC word tegen sending MAC van bericht vergeleken

-receiving AES word gebruikt om bericht de decrypten

-Als D-H key van Alice overeenkomt met laatste die je kent, zet ‘m als vorig d-h key en gebruik de nieuwe die ze hebben meegestuurd als nieuwste.

Bron: <https://otr.cypherpunks.ca/Protocol-v3-4.0.0.html>

# OTR deel 2

Off-the-Record Messaging (OTR) is a cryptographic protocol that provides encryption for instant messaging conversations. OTR uses a combination of AES symmetric-key algorithm with 128 bits key length, the Diffie–Hellman key exchange with 1536 bits group size, and the SHA-1 hash function. In addition to authentication and encryption, OTR provides forward secrecy and malleable encryption.

**Implementation**

In addition to providing encryption and authentication — features also provided by typical public-key cryptography suites, such as [PGP](https://en.wikipedia.org/wiki/Pretty_Good_Privacy" \o "Pretty Good Privacy), [GnuPG](https://en.wikipedia.org/wiki/GNU_Privacy_Guard" \o "GNU Privacy Guard), and [X.509](https://en.wikipedia.org/wiki/X.509" \o "X.509) ([S/MIME](https://en.wikipedia.org/wiki/S/MIME" \o "S/MIME)) — OTR also offers some less common features:

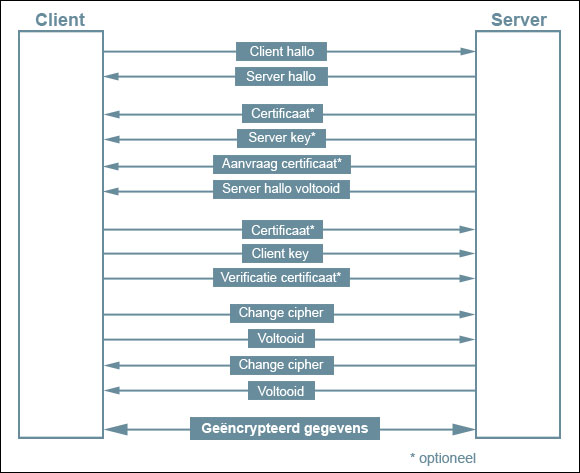
* [Forward secrecy](https://en.wikipedia.org/wiki/Forward_secrecy" \o "Forward secrecy): Messages are only [encrypted](https://en.wikipedia.org/wiki/Encrypted" \o "Encrypted) with temporary per-message [AES](https://en.wikipedia.org/wiki/Advanced_Encryption_Standard" \o "Advanced Encryption Standard) keys, negotiated using the [Diffie-Hellman key exchange](https://en.wikipedia.org/wiki/Diffie-Hellman_key_exchange" \o "Diffie-Hellman key exchange) protocol. The compromise of any long-lived cryptographic keys does not compromise any previous conversations, even if an attacker is in possession of [ciphertexts](https://en.wikipedia.org/wiki/Ciphertext" \o "Ciphertext).
* [Deniable authentication](https://en.wikipedia.org/wiki/Deniable_authentication" \o "Deniable authentication): Messages in a conversation do not have [digital signatures](https://en.wikipedia.org/wiki/Digital_signature" \o "Digital signature), and after a conversation is complete, anyone is able to forge a message to appear to have come from one of the participants in the conversation, assuring that it is impossible to prove that a specific message came from a specific person. Within the conversation the recipient can be sure that a message is coming from the person they have identified.

**Authentication**

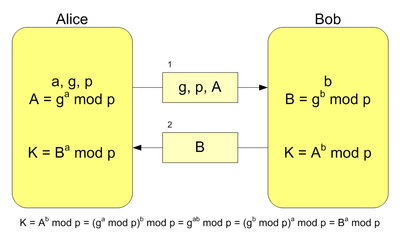
As of OTR 3.1, the protocol supports mutual authentication of users using a shared secret through the [socialist millionaire](https://en.wikipedia.org/wiki/Socialist_millionaire" \o "Socialist millionaire) protocol. This feature makes it possible for users to verify the identity of the remote party and avoid a [man-in-the-middle attack](https://en.wikipedia.org/wiki/Man-in-the-middle_attack" \o "Man-in-the-middle attack) without the inconvenience of manually comparing [public key fingerprints](https://en.wikipedia.org/wiki/Public_key_fingerprint" \o "Public key fingerprint) through an outside channel.

# TLS Protocol

TLS Protocol gebruikt certificaten om de uitgewisselde gegevens te authenticeren en privacy te garanderen. Elk certificaat bevat een publieke sleutel. De eigenaar van het certificaat bezit een privésleutel die geassocieerd is met de publieke sleutel in het certificaat. Omdat de voor cryptografie gebaseerd op de publieke sleutel relatief veel rekentijd nodig is, gebruikt TLS Protocol een session key. Deze wordt gebaseerd op de publieke sleutel en een willekeurig getal. Dit willekeurige getal wordt uitgewisseld in het eerste bericht van het protocol (Client hallo en Server hallo). In de vervolgcommunicatie wordt vervolgens de afgeleide session key gebruikt, wat minder rekentijd kost.



Het diffie-hellman-sleuteluitwisselingsprotocolis een cryptografisch protocol, waarmee twee deelnemers die verder niets van elkaar weten over een onbeveiligd communicatiekanaal een geheime encryptiesleutel kunnen uitwisselen, die daarna kan worden gebruikt om communicatie tussen de deelnemers te versleutelen.



Het idee achter dit algoritme is dat iemand mee kan luisteren en elke boodschap tussen Alice en Bob onderschept kan worden. De kunst is dus om de boodschappen zo te formuleren dat alleen Alice en Bob weten waar het over gaat. Hiervoor moeten zij vooraf een onderlinge versleuteling afspreken. Het Diffie-Hellman sleutelwisselingsprotocol maakt dit mogelijk zonder dat meeluisteraars de versleuteling kunnen onderscheppen.

De volgende stappen worden uitgevoerd:

1. Alice bedenkt een priemgetal p en een basisgetal g.  
2. Alice verzint een geheim getal a. Vervolgens berekent ze op basis van a, p en g het getal A.  
A = g^a \mod  p  
3. Alice verstuurt p, g en A naar Bob. 4. Bob verzint ook een geheim getal b en berekent op basis van de ontvangen p en g op dezelfde manier het getal B dat hij naar Alice stuurt.  
B = g^b \mod  p  
4. Bob berekent op basis van A,b en p het gemeenschappelijke geheime getal X X = A^b \mod  p  
5. Alice berekent eveneens het gemeenschappelijke geheime getal X, maar dan op basis van B, a en p. X = B^a \mod  p

De getallen p, g, A en B zijn publiek, wat wil zeggen dat iedereen die meeluistert, deze getallen kan onderscheppen. Maar het gemeenschappelijke getal dat Alice en Bob hebben berekend, is geheim, want om dat te berekenen is a of b nodig. Dit getal kunnen ze dus gebruiken om hun berichten naar elkaar toe te versleutelen. Voor een buitenstaander is het zeer moeilijk om X te achterhalen zonder a of b, want er zijn namelijk vele manieren om met p en g tot A of B te komen.

Hieronder het bewijs dat Alice en Bob tot een gemeenschappelijk getal X komen.  
A = g^a \mod  p  
B = g^b \mod  p  
X = (g^a \mod  p)^b \mod  p = g^{ab} \mod  p = (g^b \mod  p)^a \mod  p

# AES (Rijndael)

In de cryptografie is Advanced Encryption Standard (AES) een computerversleutelingstechniek (encryptie). Het is de opvolger van de "Data Encryption Standard" (DES).

AES is een subset van het Rijndael-algoritme waarbij de blokgrootte 128-bits is, en de sleutel 128, 192 of 256 bits. Rijndael zelf kan alle blokgrootten en sleutels aan die een veelvoud zijn van 32-bit met een minimum van 128-bit en een maximum van 256-bit

**Rijndael**

**Werking**

Om Rijndael toe te passen wordt eerst de te vercijferen tekst in blokken opgedeeld. Deze blokken worden vervolgens in [matrixvorm](https://nl.wikipedia.org/wiki/Matrix_%28wiskunde%29) geplaatst. De grootte van deze blokken kan variëren van 128 bits tot 256 bits met een stapgrootte van 32 bits. Dit houdt in dat men de volgende blokgroottes kan krijgen:

* 128 bits (16 bytes)
* 160 bits (20 bytes)
* 192 bits (24 bytes)
* 224 bits (28 bytes)
* 256 bits (32 bytes)

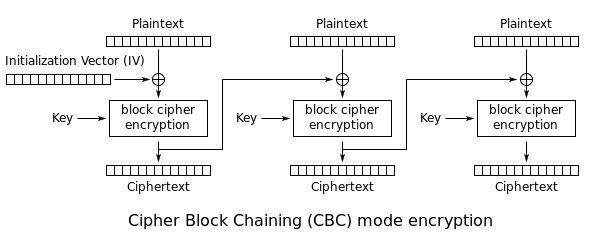
De sleutel wordt ook in blokken opgedeeld en in een [matrixvorm](https://nl.wikipedia.org/wiki/Matrix_%28wiskunde%29) geplaatst. De sleutelgrootte kan theoretisch alle sleutelgroottes aannemen van minstens 128 bits en ook een stapgrootte van 32 bits. Vervolgens wordt een aantal ronden toegepast. Het aantal ronden is afhankelijk van de lengte van de sleutel en van het blok. Het aantal rondes zou men kunnen berekenen met deze formule: y = x/32 + 6waar x de blokgrootte of sleutelgrootte is, in bits. Van de blokgrootte en sleutelgrootte wordt de grootste gekozen om x te 'vullen'. Dus bij een blokgrootte van 128 bits en een sleutelgrootte van 256 bits wordt x = 256 ingevuld en dan kan men dus het aantal rondes uitrekenen door: y = 256/32 + 6 = 8 + 6 = 14Men kan nagaan dat dan dit het aantal rondes is voor een paar andere groottes:

|  |  |  |  |
| --- | --- | --- | --- |
|  | **128 bits** | **192 bits** | **256 bits** |
| 128 bits | 10 | 12 | 14 |
| 192 bits | 12 | 12 | 14 |
| 256 bits | 14 | 14 | 14 |

Welke richting (horizontaal of verticaal) de block- of sleutelgrootte is mag u zelf weten.

AES is hetzelfde als Rijndael, maar dan alleen met de blokgrootte 128 bits en de sleutelgroottes 128 bits, 192 bits en 256 bits. Hier wordt heel vaak de fout in gemaakt.

Om een blok te vercijferen wordt eerst het blok met een [XOR](https://nl.wikipedia.org/wiki/XOR)-operatie op de sleutel verwerkt, vervolgens wordt het bovenstaand aantal ronden uitgevoerd, de laatste ronde wordt alleen de mix-columns stap overgeslagen.



# RSA is een asymmetrisch encryptiealgoritme

**Werking**

**Sleutels**

Veronderstel dat [Alice](https://nl.wikipedia.org/wiki/Alice_en_Bob) ervoor wil zorgen dat [Bob](https://nl.wikipedia.org/wiki/Alice_en_Bob) haar een persoonlijk bericht kan zenden over een onveilig medium ([telefoon](https://nl.wikipedia.org/wiki/Telefonie), [internet](https://nl.wikipedia.org/wiki/Internet), ...). Ze doet het volgende om een **publieke sleutel** en een **geheime sleutel** te maken:

1. Kies twee grote [priemgetallen](https://nl.wikipedia.org/wiki/Priemgetal) *p* ≠ *q* willekeurig en onafhankelijk van elkaar. Bereken *N* = *p \* q*.
2. Kies een geheel getal 1 < *e* < φ(*n*) , dat [relatief priem](https://nl.wikipedia.org/wiki/Relatief_priem) is t.o.v. (*p*-1)\*(*q*-1).
3. Bereken *d* zo dat *e \* d* [mod](https://nl.wikipedia.org/wiki/Modulair_rekenen) (*p*-1)\*(*q*-1) = 1
4. Vernietig alle sporen van *p* en *q*.

* (Stappen 2 en 3 kunnen uitgevoerd worden met het [uitgebreid Euclidisch algoritme](https://nl.wikipedia.org/wiki/Uitgebreid_Euclidisch_algoritme); zie [modulair rekenen](https://nl.wikipedia.org/wiki/Modulair_rekenen).)

*N* en *e* zijn de publieke sleutel, *N* en *d* zijn de geheime sleutel. Alleen *d* is dus een geheim, want *N* is bekend. Alice stuurt de publieke sleutel naar Bob (over het onveilige medium, bijvoorbeeld telefoon, internet, post), en houdt de geheime sleutel geheim.

**Versleutelen**

Veronderstel dat Bob een bericht *m* naar Alice wil zenden. Hij kent *N* en *e* (publieke sleutel), want die heeft Alice hem gezonden. Hij zet de [klare tekst](https://nl.wikipedia.org/wiki/Klare_tekst) *m* om in een getal *n* < *N*, gebruik makend van een eerder afgesproken, omkeerbaar protocol. Bijvoorbeeld, elk teken in een bericht kan worden omgezet in zijn [ASCII](https://nl.wikipedia.org/wiki/ASCII_%28Tekenset%29)-code, en de codes samengevoegd tot een enkel getal. Als het nodig is kan *m* worden opgesplitst en elk stuk afzonderlijk vercijferd. Dan berekent hij de [cijfertekst](https://nl.wikipedia.org/wiki/Cijfertekst) *c*:

 c = n^e\ \mathrm{mod}\ N

Dit kan snel gedaan worden door [machtsverheffing door kwadrateren](https://nl.wikipedia.org/wiki/Machtsverheffing_door_kwadrateren). Bob verzendt dan *c* naar Alice.

**Ontcijferen**

Alice ontvangt *c* van Bob, en kent haar geheime sleutel *d*. Ze kan *n* te weten komen uit *c* op de volgende manier:

 c^d\  \mathrm{mod}\ N = n 

Alice kan dan *n* vinden, aangezien *n* < *N*, en uit *n* kan ze dan het oorspronkelijke bericht *m* vinden.

Het ontcijferen werkt omdat

 c^d \mathrm{mod}\ N= n^{e \cdot d}\ \mathrm{mod}\ N

en *e\*d* mod (*p*-1) = 1 en *e\*d* mod (*q*-1)= 1. [Fermats kleine stelling](https://nl.wikipedia.org/wiki/Fermats_kleine_stelling) geeft:

 n^{e \cdot d} \mathrm{mod}\ p = n\  en  n^{e \cdot d} \mathrm{mod}\ q = n

en dus (aangezien *p* en *q* *verschillende* priemgetallen zijn):

 n^{e \cdot d} \mathrm{mod}\ (p \cdot q) = n 

**Voorbeeld**

Als voorbeeld kunnen we de priemgetallen 11 en 29 nemen: p=11, q=29. Nis dan 319. We kiezen e=3. 3 is relatief priem ten opzichte van (p-1)(q-1) = 10*28 = 280. dwordt dan 280-93= 187:

e * d\,\bmod\,(p-1)(q-1) \equiv 3 * 187\,\bmod\,280 \equiv 1

Als we nu de letter 'd' willen coderen (ASCII-code 100) dan krijgen we:

n^e \,\bmod\,N \equiv 100^3 \,\bmod\,319 \equiv 254

De codetekst bevat dus het karakter met de ASCII-code 254 (þ). Om de tekst vervolgens te decoderen met de geheime sleutel, krijgen we:

c^d \,\bmod\,N \equiv 254^{187} \,\bmod\,319 = 100

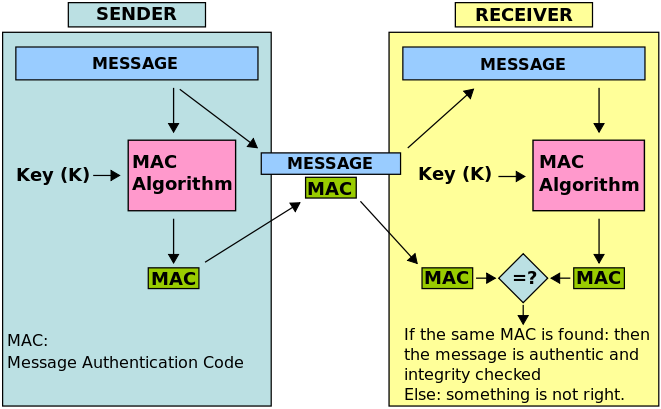
**Ondertekenen**

RSA kan ook worden gebruikt om een bericht te [ondertekenen](https://nl.wikipedia.org/wiki/Digitale_handtekening). Veronderstel dat Alice een ondertekend bericht wil zenden naar Bob. Ze berekent dan een [hashwaarde](https://nl.wikipedia.org/wiki/Hashfunctie) uit het bericht, vercijfert die met haar geheime sleutel, en voegt dat als een "handtekening" bij het bericht. Deze handtekening kan alleen worden ontcijferd met haar publieke sleutel. Wanneer Bob het ondertekende bericht ontvangt, ontcijfert hij de handtekening met Alices publieke sleutel, en vergelijkt de aldus bekomen hashwaarde met de eigenlijke hashwaarde van het bericht. Als die gelijk zijn, weet hij dat de auteur van het bericht de geheime sleutel van Alice bezit (dus normaal gezien Alice is), en dat het bericht na verzending niet meer veranderd is.

# Message authentication code

In cryptography, a message authentication code (MAC) is a short piece of information used to authenticate a message—in other words, to provide integrity and authenticity assurances on the message. Integrity assurances detect accidental and intentional message changes, while authenticity assurances affirm the message's origin.

## An Example of Message Authentication Code Algorithm

[](https://en.wikipedia.org/wiki/File:MAC.svg)

In this example, the sender of a message runs it through a MAC algorithm to produce a MAC data tag. The message and the MAC tag are then sent to the receiver. The receiver in turn runs the message portion of the transmission through the same MAC algorithm using the same key, producing a second MAC data tag. The receiver then compares the first MAC tag received in the transmission to the second generated MAC tag. If they are identical, the receiver can safely assume that the integrity of the message was not compromised, and the message was not altered or tampered with during transmission.

However, to allow the receiver to be able to detect [replay attacks](https://en.wikipedia.org/wiki/Replay_attack), the message itself must contain data that assures that this same message can only be sent once (e.g. time stamp, sequence number or use of a one-time MAC). Otherwise an attacker could — without even understanding its content — record this message and play it back at a later time, producing the same result as the original sender.

# Digital Signature Algorithm

**Key generation**

Key generation has two phases. The first phase is a choice of *algorithm parameters* which may be shared between different users of the system, while the second phase computes public and private keys for a single user.

**Parameter generation**

* Choose an approved [cryptographic hash function](https://en.wikipedia.org/wiki/Cryptographic_hash_function) *H*. In the original DSS, *H* was always [SHA-1](https://en.wikipedia.org/wiki/SHA-1), but the stronger [SHA-2](https://en.wikipedia.org/wiki/SHA-2) hash functions are approved for use in the current DSS.[[5]](https://en.wikipedia.org/wiki/Digital_Signature_Algorithm#cite_note-FIPS-186-4-5)[[9]](https://en.wikipedia.org/wiki/Digital_Signature_Algorithm#cite_note-FIPS-180-4-9) The hash output may be truncated to the size of a key pair.
* Decide on a key length *L* and *N*. This is the primary measure of the [cryptographic strength](https://en.wikipedia.org/wiki/Cryptographic_strength) of the key. The original DSS constrained *L* to be a multiple of 64 between 512 and 1024 (inclusive). NIST 800-57 recommends lengths of 2048 (or 3072) for keys with security lifetimes extending beyond 2010 (or 2030), using correspondingly longer *N*.[[10]](https://en.wikipedia.org/wiki/Digital_Signature_Algorithm#cite_note-10) FIPS 186-3 specifies *L* and *N* length pairs of (1024,160), (2048,224), (2048,256), and (3072,256).[[4]](https://en.wikipedia.org/wiki/Digital_Signature_Algorithm#cite_note-FIPS-186-3-4)
* Choose an *N*-bit prime *q*. *N* must be less than or equal to the hash output length.
* Choose an *L*-bit prime modulus *p* such that *p*–1 is a multiple of *q*.
* Choose *g*, a number whose [multiplicative order](https://en.wikipedia.org/wiki/Multiplicative_order) modulo *p* is *q*. This may be done by setting *g* = *h*(*p*–1)/*q* mod *p* for some arbitrary *h* (1 < *h* < *p*−1), and trying again with a different *h* if the result comes out as 1. Most choices of *h* will lead to a usable *g*; commonly *h*=2 is used.

The algorithm parameters (*p*, *q*, *g*) may be shared between different users of the system.

**Per-user keys**

Given a set of parameters, the second phase computes private and public keys for a single user:

* Choose *x* by some random method, where 0 < *x* < *q*.
* Calculate *y* = *gx* mod *p*.
* Public key is (*p*, *q*, *g*, *y*). Private key is *x*.

There exist efficient algorithms for computing the [modular exponentiations](https://en.wikipedia.org/wiki/Modular_exponentiation) *h*(*p*–1)/*q* mod *p* and *gx* mod *p*, such as [exponentiation by squaring](https://en.wikipedia.org/wiki/Exponentiation_by_squaring).

**Signing**

Let Hbe the hashing function and mthe message:

* Generate a random per-message value kwhere 0 < k < q
* Calculate r=\left(g^{k}\bmod\,p\right)\bmod\,q
* In the unlikely case that r=0, start again with a different random k
* Calculate s=k^{-1}\left(H\left(m\right)+xr\right)\bmod\,q
* In the unlikely case that s=0, start again with a different random k
* The signature is \left(r,s\right)

The first two steps amount to creating a new per-message key. The modular exponentiation here is the most computationally expensive part of the signing operation, and it may be computed before the message hash is known. The modular inverse k^{-1}\bmod\,qis the second most expensive part, and it may also be computed before the message hash is known. It may be computed using the [extended Euclidean algorithm](https://en.wikipedia.org/wiki/Extended_Euclidean_algorithm) or using [Fermat's little theorem](https://en.wikipedia.org/wiki/Fermat%27s_little_theorem) as k^{q-2}\bmod\,q.

**Verifying**

* Reject the signature if 0<r<qor 0<s<qis not satisfied.
* Calculate w=s^{-1}\bmod\,q
* Calculate u_{1}=H\left(m\right)\cdot w\,\bmod\,q
* Calculate u_{2}=r\cdot w\,\bmod\,q
* Calculate v=\left(\left(g^{u_{1}}y^{u_{2}}\right)\bmod\,p\right)\bmod\,q
* The signature is invalid unless v=r

DSA is similar to the [ElGamal signature scheme](https://en.wikipedia.org/wiki/ElGamal_signature_scheme).

**Correctness of the algorithm**

The signature scheme is correct in the sense that the verifier will always accept genuine signatures. This can be shown as follows:

First, if *g* = *h(*p*− 1)/*q mod *p* it follows that *gq* ≡ *hp*− 1 ≡ 1 (mod *p*) by [Fermat's little theorem](https://en.wikipedia.org/wiki/Fermat%27s_little_theorem). Since *g* > 1 and *q* is prime, *g* must have order *q*.

The signer computes

s=k^{-1}(H(m)+xr)\bmod\,q

Thus


\begin{align}
k & \equiv H(m)s^{-1}+xrs^{-1}\\
  & \equiv H(m)w + xrw \pmod{q}
\end{align}


Since *g* has order *q* (mod p) we have

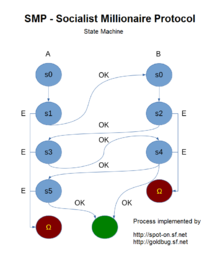

\begin{align}
g^k & \equiv g^{H(m)w}g^{xrw}\\
    & \equiv g^{H(m)w}y^{rw}\\
    & \equiv g^{u1}y^{u2} \pmod{p}
\end{align}


Finally, the correctness of DSA follows from

\begin{align}
 r &= (g^k \bmod\,p) \bmod\,q\\
   &= (g^{u1}y^{u2} \bmod\,p) \bmod\,q\\
   &= v
\end{align}

# SMP Socialist millionaire

## Off The Record Messaging protocol

[](https://en.wikipedia.org/wiki/File:SMP_-_Socialist_Millionaire_Protocol.png)

State Machine Process for the Socialist Millionaire Protocol (SMP) implemented by GoldBug.sf.net Instant Messenger (<http://goldbug.sf.net>) and Spot-On Applikation (<http://spot-on.sf.net>).

A prime, \scriptstyle p, and any non-identity element, \scriptstyle h, of \scriptstyle (\mathbb{Z}/p\mathbb{Z})^*are agreed on before the protocol, and in practice are generally fixed in a given implementation. For example, in the [Off-the-Record Messaging](https://en.wikipedia.org/wiki/Off-the-Record_Messaging) protocol, \scriptstyle pis a specific fixed 1,536-bit prime. \scriptstyle his then a generator of \scriptstyle (\mathbb{Z}/p\mathbb{Z})^*, and all operations are performed modulo \scriptstyle p, or in other words, in the [multiplicative group](https://en.wikipedia.org/wiki/Multiplicative_group), \scriptstyle (\mathbb{Z}/p\mathbb{Z})^*.

By \scriptstyle \langle h|a,\,b\rangle, denote the [secure multiparty computation](https://en.wikipedia.org/wiki/Secure_multiparty_computation), [Diffie–Hellman–Merkle key exchange](https://en.wikipedia.org/wiki/Diffie%E2%80%93Hellman%E2%80%93Merkle_key_exchange#Generalization_to_finite_cyclic_groups), which, for the integers, \scriptstyle a,\, b, returns \scriptstyle h^{ab}to each party:

* Alice calculates \scriptstyle h^aand sends it to Bob, who then calculates \scriptstyle \left(h^a\right)^b ~\equiv~ h^{ab}.
* Bob calculates \scriptstyle h^band sends it to Alice, who then calculates \scriptstyle \left(h^b\right)^a ~\equiv~ h^{ba}.

\scriptstyle h^{ab} ~\equiv~ h^{ba}as \scriptstyle (\mathbb{Z}/p\mathbb{Z})^*is Abelian. Note that this procedure is insecure against [man-in-the-middle](https://en.wikipedia.org/wiki/Man-in-the-middle) attacks.

The Socialist millionaire protocol only has a few steps that are not part of the above procedure, and the security of each relies on the difficulty of the [discrete logarithm](https://en.wikipedia.org/wiki/Discrete_logarithm) problem, just as the above does. All sent values also include zero-knowledge proofs that they were generated according to protocol.

Part of the security also relies on random secrets. However, as written below, the protocol is vulnerable to poisoning if Alice or Bob chooses any of \scriptstyle a, \scriptstyle b, \scriptstyle \alpha, or \scriptstyle \betato be zero. To solve this problem, each party must check during the [Diffie-Hellman](https://en.wikipedia.org/wiki/Diffie-Hellman) exchanges that none of the \scriptstyle h^a,\, h^b,\, h^\alpha,or \scriptstyle h^\betathat they receive is equal to 1. It is also necessary to check that \scriptstyle P_a ~\neq~ P_band \scriptstyle Q_a ~\neq~ Q_b.

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Alice** | **Multiparty** | **Bob** |
| 1 | Message x Random a, \alpha, r | Public p, h | Message y Random b, \beta, s |
| 2 |  | Secure g=\langle h|a, b\rangle |  |
| 3 |  | Secure \gamma=\langle h|\alpha, \beta\rangle |  |
| 4 | Test h^b \neq 1, h^\beta \neq 1 |  | Test h^a \neq 1, h^\alpha \neq 1 |
| 5 | \begin{align}   P_a &= \gamma^r \\   Q_a &= h^r g^x \end{align} |  | \begin{align}   P_b &= \gamma^s \\   Q_b &= h^s g^y \end{align} |
| 6 |  | Insecure exchange P_a, Q_a, P_b, Q_b |  |
| 7 |  | Secure c = \left\langle\left. Q_aQ_b^{-1} \right| \alpha, \beta \right\rangle |  |
| 8 | Test P_a \neq P_b, Q_a \neq Q_b |  | Test P_a \neq P_b, Q_a \neq Q_b |
| 9 | Test c = P_aP_b^{-1} |  | Test c = P_aP_b^{-1} |

Note that:

\begin{align}
  P_aP_b^{-1} &= \gamma^r \gamma^{-s} = \gamma^{r - s} \\
              &= h^{\alpha\beta(r - s)}
\end{align}

and therefore

\begin{align}
  c &= \left(Q_aQ_b^{-1}\right)^{\alpha\beta} \\
    &= \left(h^r g^x h^{-s} g^{-y}\right)^{\alpha\beta} 
     = \left(h^{r - s} g^{x - y}\right)^{\alpha\beta} \\
    &= \left(h^{r - s} h^{ab(x - y)}\right)^{\alpha\beta}
     = h^{\alpha\beta(r - s)} h^{\alpha\beta ab(x - y)} \\
    &= \left(P_aP_b^{-1}\right) h^{\alpha\beta ab(x - y)}
\end{align}.

Because of the random values stored in secret by the other party, neither party can force \scriptstyle cand \scriptstyle P_aP_b^{-1}to be equal unless \scriptstyle xequals \scriptstyle y, in which case \scriptstyle h^{\alpha\beta ab(x - y)} ~=~ h^0 ~=~ 1. This proves correctness.