Efficient Secure Transport

denis bider, January – April 2014

June 2019:
EXAMPLE PROTOCOL
FOR DISCUSSION

DO NOT IMPLEMENT

To Jana and Aaron :-)

1 Summary

This document defines EST, which I hypothesize is an efficient and secure internet transport protocol which operates on top of UDP, and is intended to provide a suitable replacement for TCP, SSH2, and TLS. The EST protocol is an attempt to integrate lessons learned from predecessor protocols in order to provide a unified solution that attempts to be more efficient, more secure, and easier to implement.

1.1 Motivation

This protocol attempts to avoid what I perceive as the following flaws in currently existing protocols:

1.1.1 Performance

On the current internet, TCP tends to achieve transfer speeds several times lower than what is actually supported by a given connection. This is due to its conservative reaction when faced with any data loss, and slow congestion window growth in networks with a high bandwidth-delay product. Our current internet is a consumer content delivery platform where speed is important, but tends to have roundtrip times higher than TCP implementations are good at. The ubiquity of wireless connections means a small amount of data loss must be expected and tolerated without a dramatic deterioration of performance.

EST is based on UDP instead of TCP[1], implements congestion control that is resilient to accidental packet loss, and makes better use of network paths with a high bandwidth-delay product.

1.1.2 Resilience

SSH2 and TLS are not resilient to data corruption and external obstruction on the TCP layer. Data corruption can be accidental: when large amounts of data are being transferred over a connection prone to data distortion, eventually a distorted datagram will pass the 2-byte TCP checksum, and cause a terminated SSH2/TLS session. Sessions can also be terminated easily by attackers, who only need to impersonate the IP address of one of the parties, and send a single RST packet to the other.

EST is resilient to this by using UDP, and discarding datagrams that do not pass message authentication.

1.1.3 Central Authorities

The use of X.509 and the encouragement of global root certificate authorities in TLS was a likely security blunder, though a necessary one since it was designed at a time when Bitcoin didn't yet exist. The blockchain mining concept introduced by Bitcoin solves the problem of peer-to-peer name registration, which means the internet can migrate to this decentralized solution, and away from root DNS servers and certificate authorities, the latter of which have been shown vulnerable to hacking, as well as abuse by national authorities.

[1] From a performance standpoint, it would have been marginally better to design EST directly on top of IP, but the main savings would be a few bytes per datagram for UDP checksum and port numbers, and it would deter widespread use: protocols other than TCP and UDP are blocked by many networks, and often cannot be implemented by non-privileged applications.

EST takes a lesson from SSH, and uses raw public keys to authenticate servers. Before connecting to the server, the client is expected to obtain a fingerprint of the server's public key using a trusted mechanism external to EST. In separate work, I describe a mechanism to provide this service using Bitcoin.

1.1.4 ASN.1

A lesson learned from TLS and X.509 is that ASN.1 is hard for most people to implement correctly, and should be avoided. EST tolerates the use of ASN.1 for representation of e.g. key data which would normally be handled by well-written cryptographic libraries. The rest of the protocol, however, uses an easier to implement encoding inspired by SSH and Bitcoin.

1.1.5 Key Exchange

SSH2 key exchange negotiates separate encryption and MAC algorithms for client-to-server and server-to-client directions. In practice, implementations negotiate the same algorithms for both directions, and the extra negotiation is useless. EST uses the same set of negotiated algorithms for both directions, and negotiates encryption and MAC algorithms as one, since they are not always orthogonal (e.g. AES-GCM).

1.1.6 Key Re-Exchange

SSH2 key re-exchange is difficult to implement correctly. It imposes a computational cost, as well as a short, but noticeable period of congestion on what could otherwise be a smooth, flowing connection. EST avoids the need for key re-exchange, and provides seamless rollover, by relying on a single shared secret to seed an endless supply of symmetric keys.

1.1.7 Public Key Algorithms

The way SSH2 specifies public key algorithms used in server authentication requires clients to verify or import each key separately, which makes it challenging for a server to change to a different host key algorithm. EST implements mechanisms allowing an implementation that trusts one fingerprint of the other party to automatically acquire the party's other fingerprints during a session.

1.1.8 Public Key Rollover

For security, it is necessary to roll over public/private keypairs from time to time. However, the SSH2 algorithm provides no mechanism to facilitate this automatically. A change of host keys means all clients have to re-verify/re-import the new host key, which often prohibits key rollover, or delays it considerably.

EST both provides mechanisms to automate key rollover during the natural course of key exchange, and reduces the need for it by using sub-certificates to minimize the number of times a master key is used.

1.1.9 Channels

The SSH2 session is designed to multiplex one or more channels, which can be a privacy boon: it is harder to analyze traffic when independent data sent on separate channels is indistinguishable to the attacker. However, it is a curse with respect to performance and implementation complexity. It requires duplicating TCP flow control on the level of individual channels. Maintaining channel responsiveness is challenging when another channel is highly active, and the underlying transport is buffered.

Due to its use of individual datagrams instead of serializing channel data into a master stream, EST can provide the best of both worlds: the improved privacy of indistinguishable multiple channels, and independent channel flow control without the stream serialization penalty.

1.1.10 Versatility

Existing transport security protocols, such as TLS and SSH, are inappropriate for use in datagram-driven applications. Their design and performance are unsuitable for applications such as games or UDP tunneling. EST does not have this problem, and can encrypt datagram-based traffic while preserving the performance characteristics (and lack of delivery guarantees) of UDP.

1.1.11 Client Authentication

Previous security-minded protocols, such as TLS and SSH, were shy when it came to mandating public keys for client authentication, and left this as a decision for individual users. Since most end-users lack expertise with and understanding of authentication, most client authentication over TLS and SSH continues to be performed with passwords that are predominantly of poor quality, while websites continue to store passwords in databases that are prone to being stolen, and easily cracked.

EST addresses this by mandating public key authentication for all clients. This is an attempt to herd client software implementors towards improving the end-user's public key authentication experience, so that it becomes easy, intuitive, and commonplace.

2 Standardization

June 2019: The original content here was a matter of opinion that no longer reflects my present views. It would detract from discussion, so it was removed.

3 Definitions

3.1 Terms

Session	An entire EST exchange starting from the first DTG_INIT until the last DTG_ENVELOPE.
Key exchange	The DTG_INIT/REPLY and DTG_KEX/REPLY datagrams at the start of a session.
Connection	The part of session after a successful key exchange. Uses DTG_ENVELOPE datagrams.

3.2 Encoding

The EST protocol uses the following encoding primitives. No alignment or padding bytes should be implied, unless explicitly noted:

`byte` — An octet.

`type``[n]` — *n* values of type *type* encoded serially one after another.

`esthash` — Another name for *byte[20]*. See EST-HASH.

`uint16` — A 16-bit unsigned integer, encoded **most** significant byte first.

`varuint` — A variable-length unsigned integer, encoded as follows:

Min value	*Max value*	*Mod*	*N*	*Test*
0	0x7F	f |= 0x00	1	(f & 0x80) == 0
0x80	0x3FFF	f |= 0x80	2	(f & 0x40) == 0
0x4000	0x1FFFFF	f |= 0xC0	3	(f & 0x20) == 0
0x200000	0x0FFFFFFF	f |= 0xE0	4	(f & 0x10) == 0
0x10000000	0x07FFFFFFF	f |= 0xF0	5	(f & 0x08) == 0
0x0800000000	0x03FFFFFFFFF	f |= 0xF8	6	(f & 0x04) == 0
0x040000000000	0x01FFFFFFFFFFF	f |= 0xFC	7	(f & 0x02) == 0
0x02000000000000	0x00FFFFFFFFFFFFF	f |= 0xFE	8	(f & 0x01) == 0
0x0100000000000000	0xFFFFFFFFFFFFFFFF	pre = 0xFF	9	else

The **most** significant non-zero byte of the value, *f*, is modified as specified in the *"Mod"* column; the result stored as the first byte. In the case of N = 9, the byte 0xFF is prepended without modifying the value. Bytes representing the rest of the value are then appended, most significant byte first. *N* is the total number of bytes encoded.

To decode, test the first encoded byte, *f*, against rules in the *Test* column to determine the number of bytes that comprise the value. Tests **must** be performed in sequence. Once the right *N* is found, strip the *"Mod"* bits, and decode the rest of the value.

An instance of encoding that uses more bytes than required is **non-canonical**. Encoders **must not** encode a non-canonical value, and decoders **must not** accept them.

`nstime`	A fixed-width 64-bit **unsigned** integer, encoded **most** significant byte first, representing a time such that the value is the number of **nanoseconds** since 00:00:00.000000000 UTC on January 1, 1970, not counting leap seconds. Equivalent to Unix time, except in nanoseconds. This is sufficient to encode points in time from 1970 until around 2554.
`vect<type>`	Encoded as follows: varuint n Number of values. *type*[n] values A series of *n* encoded values.
`string`	Another name for *vect<byte>*.
`stdname`	A string of 1 or more bytes, each of which matches a set of characters consisting of [a-z], lowercase only; [0-9]; and dash (-). The first character must be in [a-z], the last character must not be a dash, and there may not be two consecutive dashes.
`domain`	A string of two or more *stdname* separated by a single period (.). There may not be a leading period, a trailing period, and there may not be two consecutive periods.
`name`	A non-empty *string* matching exactly one of the following: 1. A *stdname*. **2.** A **private name**, which is a *stdname* representing the algorithm, followed by the ASCII character 64 (@), followed by a *domain* representing the private domain namespace.
`namelist`	A *string* matching exactly one of the following: 3. Empty string. 4. A *name*. 5. A list of two or more *name*s, with each non-first name separated from its predecessor by exactly one comma. There may be no trailing or leading commas, and no whitespace. The names appearing in the list may be either standard or private.

4 Sessions and Ports

An EST session has two participants, a server and a client. The server accepts UDP datagrams on an interface and port number known to legitimate clients. The client send UDP datagrams from an ephemeral port number, which the server replies to.

4.1 Port Unpredictability

To impede the ability of attackers to mount large-scale drive-by attacks, this specification does not select a well-known port for EST connections. Other protocols based on EST are likewise advised not to select a well-known port. Individual server implementations are advised to make the port number configurable by an administrator, and to use an installation-specific random port number by default.

Mechanisms designed for EST server lookup are advised not to provide mere lookup of the form:

```
(name) -> (IP address)
```

but instead to provide the following complete information:

```
(name, service) -> (IP address, port number, server public key fingerprints)
```

4.2 Port and Address Flexibility

EST sessions are intended to be resilient to a client's **or** server's change of IP address and port number. This allows an EST session to continue through router resets, or switching internet connections. An implementation should always send a reply datagram to that IP address and port number from which the last valid datagram in the session was received. The last valid message is the one with the highest datagram type ID (first byte of datagram) during key exchange, representing furthest progress; or after key exchange, the DTG_ENVELOPE datagram with the highest transmission sequence number.

4.3 Connection Lifetime

Servers **must** keep connection state, and allow continuation of a session, for at least 150 seconds after the last valid datagram was received from the client.

Clients **should** attempt to reuse an existing EST session that matches the desired session parameters, as long as the last valid datagram sent by the client as part of that session was less than 120 seconds ago.

EST does provide MSG_ABORTSESSION, but this should **not** be used as part of the normal course of communication. MSG_ABORTSESSION should be sent only by a party which expects to no longer be able to handle EST sessions, either in general because it's e.g. shutting down, or with that specific remote party, because it was e.g. administratively banned.

As long as a process continues to be ready to handle EST traffic with a party it's communicating with, it should **not** abort a recently active connection. If the connection has an inactive channel, and a party wants to clean up resources associated with that channel, the party **may** signal abandonment of that channel using the CHANFL_CLOSED flag in MSG_STAT, but should **not** send MSG_ABORTSESSION.

5 Resilience Against Spoofing

"Spoofing" describes attacks where an uninvited party sends datagrams that appear to originate from a different location than their real sender. Two types of spoofing are considered in this section:

1. **Basic-knowledge** spoofing, where the attacker can know or predict the IP addresses and port numbers of the parties involved in an EST connection, but does not have immediate access to the content of datagrams exchanged between the two parties at the time of the attack.
2. **Advanced-knowledge** spoofing, where the attacker has immediate access to the contents of some or all datagrams exchanged between the parties at the time of the attack.

In EST, the effect of any successful spoofing attack is limited to denial of service. A successful spoofing attack during key exchange would prevent the key exchange from taking place, while a successful spoofing attack after key exchange would disrupt the connection.

Both client and server EST implementations **must** be resilient to basic and advanced-knowledge spoofing at all times. This makes EST resistant to all interference except by an attacker who can reliably prevent datagrams from being delivered between the two parties – in which case, the attacker is able to disrupt communication regardless of the protocol in use.

5.1 Implementation

For protection against advanced-knowledge spoofing after key exchange, clients and servers **must** silently discard any datagrams that are not DTG_ENVELOPE, and that do not contain a valid *remoteSessionId* field and a valid integrity envelope. To protect against replay attack, DTG_ENVELOPE datagrams with a valid integrity envelope, but with a duplicate *xmitNr*, **must** also be ignored.

For protection against advanced-knowledge spoofing during key exchange:

- Servers **must** silently discard DTG_KEX packets that do not have a valid *serverSessionId* field, or a valid signature corresponding to a client public key fingerprint in DTG_INIT.
- Clients must assume that **any** datagram received is spoofed, unless it is a valid member of a chain of datagrams leading up to a valid DTG_KEXREPLY. Before a valid DTG_KEXREPLY is received, any preceding DTG_INITREPLY could have been spoofed. Therefore, until a valid DTG_KEXREPLY is received, the client **must** continue resending its original DTG_INIT, followed by a separate DTG_KEX response to each unique, potentially valid DTG_INITREPLY the client has received so far.

6 Algorithm Negotiation

Algorithm negotiation between the client and the server is done using strings of type *namelist*. Each such string is a list of mutually exclusive algorithm options supported by the side that sends the list. The client **should** organize the name list in the order of the user's preference. The server may organize the name list in any order. The algorithm chosen is the first name on the client's list that **exactly matches** (case sensitive) a name on the server's list.

If a client or server implementation supports more than one algorithm in a name list, it **must** provide a way for the user (or server administrator) to disable any of the algorithms, which must prevent them from being negotiated, or appearing in the list.

Due to the limited size of DTG_INITREPLY, and due to the accumulating nature of algorithm lists – it is more likely that an implementation will add support for a new algorithm, than remove support for an existing one – those who specify new algorithms are **encouraged** to select concise algorithm names.

6.1 Private Algorithm Names

All algorithms defined by implementations outside of the EST standardization body **must** use a private algorithm name in the *name@domain* format. The *domain* part **must** be a registered domain name associated with the implementation defining the algorithm. The full *name@domain* algorithm name is **not** recommended to be a valid email address (unless its owner wishes to attract junk mail).

6.2 Invalid Names

Implementations **may** ignore, and discard as invalid, INIT and INITREPLY datagrams containing algorithm names that do not follow the *namelist* format.

6.3 Extensions

Extension negotiation uses name lists in the same format as algorithm negotiation, but matches them differently. After DTG_INITREPLY is sent and received, the extensions that are enabled for the session are all that appear on both the server's and the client's extension list. If any subset of extensions are mutually exclusive, the one that takes precedence is the one that appears first on the client's list.

7 Server and Client Authentication

In a departure from SSH and TLS, EST mandates public key authentication during key exchange not only for the server, but also for the client. Requiring the client to use a public key during key exchange has the following advantages:

- Servers can reject unknown clients with little vulnerability exposure and computational effort.
- Servers can authenticate a client's DTG_KEX against the preceding DTG_INIT, simplifying protection against an advanced-knowledge spoofing attack.
- Applications built on EST can rely on clients to use secure authentication by default. This can be a boon for websites, which can rely on EST to authenticate users easily and securely. Users are no longer required to maintain collections of passwords for different sites.

The anonymity of clients connecting to public servers is not compromised – anonymity-concerned clients should generate a separate keypair for each website they communicate with, and for servers that do not require accounts, such keypairs can be ephemeral. However, there is important benefit to that a lack of authentication is made to be an explicit choice, rather than the default rule.

7.1 Generic Public Key Format

An EST public key is contained in a *PubKey* type, which is encoded as follows:

`name`	`algorithm`	The name of the public key algorithm.
`string`	`publicKey`	The public key encoded in an algorithm-specific manner.

7.2 Sub-Certificates

To reduce the need for master key rollover, each EST party authenticates itself using a chain of public keys. A chain may contain between 1 and 7 *SubCerts*. This allows a master key to parent over 2^53 signatures even with NTRU, where keys must be replaced every 107 signatures. Other algorithms, including those which are not believed to leak data in signatures, are still advised to use deep *SubCert* chains because most algorithms do leak data in side channels. The *SubCert* type is encoded as follows:

`string`	`publicKey`	The public key encoded in an algorithm-specific manner. Not a *PubKey* type, but the *publicKey* field from inside a *PubKey* type.
`nstime`	`expiration`	The point in time when the signature in this *SubCert* expires.
`string`	`signature`	A signature of *publicKey* and *expiration* fields as encoded, using the private key corresponding to the previous sub-certificate, or using the master private key if this is the first sub-certificate.

All *SubCert*s presented during EST key exchange **must** use the negotiated public key algorithm.

7.3 Server Identity

Servers **may** represent multiple identities at the same IP address and port. For example, a server might serve web pages for many domains with different owners. EST does not communicate during key exchange the identity the client wishes to connect to; doing so would reveal the identity in plaintext. Therefore, owners of DNS names hosted at a third party server need to ensure that their DNS names are publicly associated with the fingerprints of public keys used by said server.

When a client verifies a server's public key fingerprint, it **should** determine the fingerprint record to validate against based on the concept of identity most meaningful to the client. A client might associate server identity with the DNS name it believes it's connecting to. Trust for server key fingerprints should **not** be locked in with incidental and ephemeral data, such as IP addresses and port numbers.

7.4 Key Rollover

DTG_INIT and DTG_INITREPLY contain fields *clientKeyFps* and *serverKeyFps*, respectively. These fields contain a list of binary fingerprints, one for each master public key that the respective party uses now, or may use in the future. One of the keys referenced by each field is used in a successful key exchange. After key exchange, each party **should** store fingerprints from the other party that it didn't previously know about, and mark them as trusted, as long as they don't already appear on the party's revoked list.

The same datagram also contain fields *revokedClientFps* and *revokedServerFps*, respectively. If one of these fields is non-empty, the receiving party **must** remove any fingerprints contained there from its list of trusted fingerprints. Further, all listed fingerprints must be added to a separate revoked list, which **must** prevent such fingerprints from being trusted in the future.

7.5 Fingerprints

A public key fingerprint is calculated as EST-HASH of a *PubKey*; that is to say, over the full encoded *algorithm* and *publicKey* fields, including their lengths; but not including the length of any encapsulating *string* in which the *PubKey* might be contained. When conveyed by machines, for the purposes related to this protocol, a fingerprint is stored as a fixed-size array of 20 bytes.

When presented for human consumption, fingerprints **must** be presented to the user in **Bubble Babble** encoding. A fingerprint presented to a user will primarily be used for key verification. To facilitate key verification across varied implementations, all fingerprints must be displayed in Bubble Babble.

The *Bubble Babble Binary Data Encoding* specification, authored by Antti Huima of then SSH Security, can be found floating on the internet[2] as:

```
draft-huima-babble-01
draft-huima-babble-01.txt
```

[2] This can serve as another example of IETF's failure. A genuinely useful draft, with multiple independent implementations, never made it as an RFC – a "Request for Comments" – and must be hunted down on random people's websites. The widely used SFTP protocol enjoys the same status, though for additional reasons.

8 Key Exchange

8.1 DTG_INIT

The client begins an EST session by sending to the server a DTG_INIT datagram of size at least 1,450 bytes, and no more than 4,050 bytes. The client keeps resending the same identical DTG_INIT datagram at regular intervals until the key exchange completes with a valid DTG_KEXREPLY, or the client gives up:

byte	DTG_INIT	The value 0x30.
byte[3]	protoId	The three bytes "EST", in this sequence.
string	clientAgent	See *Agent String* and *Agent String Format* below.
varuint	clientSessionId	Client's session ID. If the client doesn't need a session ID, it **should** generate it as a random number between 0 and 127.
vect<esthash>	clientKeyFps	Fingerprints of client keys that may be used to sign DTG_KEX. See also *Server and Client Authentication* > *Key Rollover*.
vect<esthash>	revokedClientFps	See *Server and Client Authentication* > *Key Rollover*.
vect<esthash>	acceptServerFps	A list of fingerprints of server public keys the client will accept. If empty, or the server has no matching key for the negotiated algorithm, the server uses its default key for that algorithm.
namelist	extensions	Client's supported extensions that modify the protocol.
namelist	clientPkAlgs	Algorithms supported by client for authentication of client.
namelist	serverPkAlgs	Algorithms supported by client for authentication of server.
namelist	kexAlgs	Key exchange algorithms supported by the client.
namelist	encAlgs	Encryption-and-integrity algorithms supported by the client.
byte[16]	clientNonce	Random bytes generated by the client.
byte[…]	padding	See *Padding* below.

8.1.1 Agent String

All implementations **must** encode an agent string (*clientAgent* and/or *serverAgent*) that correctly and meaningfully names the EST implementation in a way that will allow a reasonably knowledgeable human to identify it.

The agent string **may** omit the exact software version. Omitting the exact version can make it more difficult for an attacker to exploit a known vulnerability in that version. Omitting the agent name, however, does not thwart a serious attacker, who can recognize the software through supported algorithms and other idiosyncrasies. Meanwhile, the absence of human readable information frustrates investigation of reports about compatibility issues.

8.1.2 Agent String Format

The agent string consists of one or more **component strings**. If there is more than one component string, each consecutive pair is separated by exactly one slash character (/). There must **not** be a leading slash character, a trailing slash character, more than one consecutive slash character, or a slash character within a component string.

A component string consists of a **component name**, optionally followed by the colon character (:) and **component version**. The component version can be omitted, in which case the colon character must also be omitted. Component name and component version must be strings consisting of one or more characters in the set [a-z], [A-Z], [0-9], and space (US ASCII value 32).

Components **should** be listed in ascending order of separation from the underlying UDP protocol. If a higher-level application uses an EST library, the name of the EST library should appear first.

The maximum length of an agent string is 0x7F bytes (so that length is one byte). Implementations **should** discard datagrams with agent strings that are empty, too long, or contain invalid characters.

Example valid agent strings:

```
estcore:1.10.10/Mysterious Component
MysteriousLibrary/Awesome Software:9.67/PluginX
mysterious software
```

8.1.3 Padding

If the size of DTG_INIT would be less than 1,450 bytes, the *padding* field **must** be present to avoid incentivizing denial of service amplification attacks. The size of padding must be such that the total size of the DTG_INIT datagram is *at least* 1,450 bytes – the maximum size of DTG_INITREPLY.

For future extensibility, a client implementing this protocol version **must** initialize the padding, if any, to all zero values. A server implementing this protocol version **must** verify that the total length of DTG_INIT is at least as required, but **must not** verify that padding values are zero.

8.2 DTG_INITREPLY

The server responds with an identical DTG_INITREPLY to each identical instance of a valid DTG_INIT. A server must **never** send a DTG_INITREPLY of size more than 1,450 bytes.

byte	DTG_INITREPLY	The value 0x31.
varuint	clientSessionId	Client's session ID.
esthash	initHash	EST-HASH of the preceding DTG_INIT.
string	serverAgent	See *Agent String* and *Agent String Format* under DTG_INIT.
varuint	serverSessionId	Server's session ID.
vect<esthash>	serverKeyFps	See *Server and Client Authentication* > *Key Rollover*. The first fingerprint listed **must** be of the public key the server intends to use in its DTG_KEXREPLY. If the client does not trust this fingerprint, it aborts the key exchange.
vect<esthash>	revokedServerFps	See *Server and Client Authentication* > *Key Rollover*.
vect<esthash>	acceptClientFps	A list of fingerprints of client keys the server will accept. **Must** contain a subset of fingerprints in *clientKeyFps* sent by the client in DTG_INIT. If empty, the client aborts key exchange.
namelist	extensions	Server's supported extensions that modify the protocol.
namelist	clientPkAlgs	Algorithms supported by server for authentication of client.
namelist	serverPkAlgs	Algorithms supported by server for authentication of server.
namelist	kexAlgs	Key exchange algorithms supported by the server.
namelist	encAlgs	Encryption-and-integrity algorithms supported by the server.
byte[16]	serverNonce	Random bytes generated by the server.

When waiting for DTG_INITREPLY, the client **must** silently discard datagrams that are unexpected or invalid in any way. Since DTG_INITREPLY is not authenticated, the client **must** assume any invalid ones are spoofed. Among the potentially valid ones, there is no way to tell which one is ultimately valid until a successful DTG_KEXREPLY.

8.3 DTG_KEX

The client responds to a valid DTG_INITREPLY by sending to the server the following DTG_KEX:

byte	DTG_KEX	The value 0x32.
varuint	serverSessionId	Server's session ID.
esthash	initReplyHash	EST-HASH of the preceding DTG_INITREPLY.
PubKey	clientMasterKey	The client's master public key used for this datagram. **Must** match a fingerprint presented in DTG_INIT.
vect<SubCert>	clientSubCerts	See *Server and Client Authentication* > *Sub-Certificates*. At least one *SubCert* **must** be encoded. At most 7 *SubCert*s may be encoded.
string	kexPayload	A payload encoded according to the key exchange algorithm negotiated in DTG_INIT/REPLY.
string	clientSig	The client's signature of this DTG_KEX, excluding the final *signature* and *proofNonce* fields. The signature is performed using the last key in *clientSubCerts*.
byte[8]	proofNonce	See *Proof of Work* below.

The size of DTG_KEX may be no more than 4,050 bytes.

The client keeps resending the same identical DTG_INIT datagram, as well as DTG_KEX responses to all unique, potentially valid DTG_INITREPLY received so far, at regular intervals until a valid DTG_KEXREPLY is received, or the client gives up.

The server **must** silently discard any DTG_KEX datagrams that do not contain a valid *serverSessionId* or *initReplyHash*, or where the *proofNonce* doesn't validate, or where the client's signature in *clientSig* does not validate. If a received DTG_KEX datagram is invalid in some other way – but the fields enumerated in the preceding sentence are valid, including signature – then the server **may** discard with prejudice any and all further datagrams chaining back to the same DTG_INIT.

8.3.1 Proof of Work

A successful EST key exchange requires a server to expend computational resources that are small with current hardware for a single session, but significant for a busy server.

To make computational denial of service attacks more difficult, and to encourage EST clients to reuse existing sessions, an EST client must prove a degree of dedication when connecting to an EST server. To do this, the client is asked to expend computational effort of similar magnitude as the server's expected effort during key exchange. To prove expenditure of effort, the client increments a randomly generated *proofNonce*, and for each value, calculates EST-HASH over the full encoding of the DTG_KEX datagram.

The client continues this process until it finds a hash beginning in two zero bytes. The server verifies the proof by calculating the hash of the DTG_KEX datagram it receives, and checking that the first two bytes of the hash are zero.

The 16-bit proof of work threshold was determined through an empirical comparison of the performance of SHA2-256, as representative of client cost, vs. 3072-bit DH keypair generation, key agreement, and RSA signing, as a conservative representative of server cost. The measured ratio was around 16,000 dual hash operations per one DH + RSA key exchange operation.

8.4 DTG_KEXREPLY

The server responds with an identical DTG_KEXREPLY to each identical instance of a valid DTG_KEX:

byte	DTG_KEXREPLY	The value 0x33.
varuint	clientSessionId	Client's session ID.
esthash	kexHash	EST-HASH of the preceding DTG_KEX. This is the same hash that serves as client's proof of work.
PubKey	serverMasterKey	The server's master public key used for this datagram. **Must** match the fingerprint presented in DTG_INITREPLY.
vect<SubCert>	serverSubCerts	See *Server and Client Authentication* > *Sub-Certificates*. At least one *SubCert* **must** be encoded. At most 7 *SubCert*s may be encoded.
string	kexReplyPayload	A payload encoded according to the key exchange algorithm negotiated in DTG_INIT and DTG_INITREPLY.
string	serverSig	The server's signature of this DTG_KEXREPLY, excluding the final *signature* field. The signature is performed using the last key in *serverSubCerts*.

The size of DTG_KEXREPLY may be no more than 4,050 bytes.

When waiting for DTG_KEXREPLY, the client **must** silently discard any datagrams that are invalid for any reason. Further, the client **must** still remain ready to accept another DTG_INITREPLY, in case the one it did receive and act upon was spoofed. The client **must** keep sending its original DTG_INIT, as well as DTG_KEX responses to every potentially valid DTG_INITREPLY, until a valid DTG_KEXREPLY is received.

9 Connection

After a successful key exchange, all further messages are enclosed in this datagram type:

9.1 DTG_ENVELOPE

Either side can send this type of datagram to transmit part of its data stream to the other party:

byte	DTG_ENVELOPE	The value 0x34.
varuint	remoteSessionId	The other side's session ID.
varuint	xmitNr	Sender's transmission sequence number for this session. This counter starts at 1 for the first transmission, and is incremented by 1 each time a DTG_ENVELOPE datagram is sent or resent by the sender.
byte[8]	xmitNonce	See *Transmit Nonce* below.
byte[…]	encryptedData	Encrypted and authenticated datagram data. Encryption and message authentication are parameterized by the session shared secret, *xmitNr*, and *xmitNonce*.

The size of DTG_ENVELOPE may be up to 65,535 bytes.

9.1.1 Transmit Nonce

Many encryption algorithms, including the one initially defined here for use in EST, are catastrophically vulnerable to conditions which would cause different plaintexts to be encrypted using the same encryption parameters. In EST, this can happen during restoration of live virtual machine state with EST sessions in progress.

Applications that could conceivably run under a virtual machine **must** generate a random *xmitNonce* that depends not just on internal RNG state, but also on the content of the *entire* data to be encrypted, *including* padding. If the application's RNG does not provide a way to incorporate entropy, one way to generate the *xmitNonce* would be as follows:

```
timestamp = 64-bit integer with current time in as much precision as possible
xmitNonceHash = SHA2-256(concatenate(8+ RNG bytes, time (µs), padded plaintext))
xmitNonce = truncate(xmitNonceHash, 8)
```

If the application's RNG can incorporate entropy, a better way would be to provide the RNG with *timestamp* and *msgData* as new entropy immediately prior to using it to generate *xmitNonce*.

Applications that are **guaranteed** to never be run within a virtual machine – such as pure hardware implementations – **may** generate a trivial *xmitNonce*.

9.1.2 Encrypted Data

The plaintext of the *encryptedData* field is encoded as follows:

`byte`	`(batchNr << 3) | msgType`	The 3 least significant bits contain *msgType*. This determines the format of data that follows. The 5 most significant bits contain *batchNr*. This determines the sending batch this datagram is part of. See *Batch Number*.
`byte[…]`	`msgData`	Data in message-specific format. Length is deduced based on the length of *encryptedData* as a whole, and the value of the last byte of padding.
`byte[…]`	`padding`	Between 1 and 256 padding bytes, each of which **must** equal the number of padding bytes itself. If the number of padding bytes is 256, the values must be zero.

The size of padding is calculated as follows:

1. Let *padBlockSize* be the cipher block size. If this is less than 16, let *padBlockSize* be 16.
2. Let *plainSize* be the size of the message plaintext.
3. Let *basePadSize* equal *plainSize* % *padBlockSize*. If this equals zero, let *basePadSize* equal *padBlockSize.* The resulting *basePadSize* is the absolute minimum amount of padding to append. This step creates a proper plaintext size if block encryption is used, and if not, limits the ability of traffic analysis to determine exact message size.
4. Let *optSize* be the minimum of (a) the receiver's optimal incoming datagram size, and (b) our optimal outgoing datagram size.
5. If, after adding *basePadSize* bytes of padding, the encrypted size of DTG_ENVELOPE would not exceed *optSize*, let *maxEncSize* be *optSize*. Otherwise, if the message is MSG_CHANNELDATA, abort. Otherwise, if the final encrypted DTG_ENVELOPE would be larger than 65,535 bytes, abort. Otherwise, let *maxEncSize* be 65,535.
6. Determine *minPadSize* as the minimum size of padding such that the final encrypted size of DTG_ENVELOPE will be at least 256 bytes. If this is less than *basePadSize*, let *minPadSize* equal *basePadSize.*
7. Determine *maxPadSize* as the maximum size of padding such that the final encrypted size of DTG_ENVELOPE will not exceed *optSize*. If *maxPadSize* is larger than 256, set it to 256.
8. Generate *padSize* as a random value between *minPadSize* and *maxPadSize*, inclusive.
9. If (*plainSize* + *padSize*) % *padBlockSize* is not zero, reduce *padSize* so that it is.
10. The resulting *padSize* is the amount of padding to append.

9.1.3 Batch Number

As sender, each side keeps track of up to 32 batches of datagrams. The concept of batches is used to help the receiver provide useful information to the sender in its MSG_STAT replies. A sender can send a consecutive series of datagrams, all sent with the same sending rate, in the same batch. Depending on the length of the sender's batch, the receiver might send MSG_STAT replies while the batch is in progress, but **must** send a MSG_STAT after receipt of a datagram that starts a new batch. The sender begins by sending batch zero, and increments the batch number by one for each subsequent batch. When the batch number has reached 31, and is incremented again, it wraps around to zero.

9.2 MSG_DATAGRAM

MSG_DATAGRAM uses *msgType* **0x00**, with *msgData* as follows:

`name`	`service`	A well-known name for a recipient service that should handle this datagram. If the recipient does not support this service, the datagram should be silently dropped.
`byte[…]`	`content`	Service-specific datagram payload.

From an application's perspective, MSG_DATAGRAM messages are as unreliable as UDP. If the message is dropped, this is not detected, and no retransmission is attempted. The difference compared to UDP is that the message, if it does arrive, is encrypted and authenticated.

This message type is intended for use by applications such as games, and for UDP tunneling.

9.3 MSG_CHANNELDATA

MSG_CHANNELDATA uses *msgType* **0x01**, with *msgData* as follows:

varuint	channelId	The channel ID, allocated by the side that opened the channel. Channels opened by clients **must** use IDs that are even or zero. Channels opened by servers **must** use IDs that are odd. Must **not** exceed *maxInChannelId* of the party that did not open the channel.
varuint	segNr	Sender's segment sequence number for this session and channel. This counter starts at 1 for the first MSG_CHANNELDATA, and is incremented by 1 each time a new CHANNELDATA message is sent by the sender. When a previously sent CHANNELDATA message is resent, this sequence number does not change.
name	service	This field is present **only** in the direction from channel opener to responder, and then **only** in the first data message for this channel (*segNr* = 1). Then, it contains the name of the service for which this channel is intended. If the recipient does not support this service, it **should** abandon the channel, and signal this using CHANFL_CLOSED in MSG_STAT.
byte[…]	segment	A part of stream data. Empty to indicate end of data in this direction.

MSG_CHANNELDATA is intended for sending in bulk. The sender should start with a reasonable transmission rate estimate, and then adjust the send rate up or down depending on information received in MSG_STAT.

A DTG_ENVELOPE containing MSG_CHANNELDATA **may not** have size larger than the receiving party's *optInSize*.

MSG_CHANNELDATA gets retransmitted on demand based on MSG_STAT information received from the other party. When retransmitting MSG_CHANNELDATA, the message is re-encrypted with the same plaintext, but a different *xmitNr* and *xmitNonce*, and therefore different encryption parameters.

A channel is opened implicitly by the sending side allocating a channel ID, and using it to send messages on that channel. The opening side will begin by sending one or more EST_CHANNEL_DATA, and reminding the other side of the channel's existence in MSG_STAT until it is acknowledged.

From an application's perspective, a channel is a bidirectional stream with serialization and reliability characteristics similar to TCP. An EST channel differs from a TCP connection in that it is encrypted and authenticated, and aims to provide better performance over realistic connections.

A channel is closed when both sides have sent and received an empty *segment* indicating end of data. Once a side knows that the other side has both sent and received an empty *segment* (via MSG_STAT), it can reuse the channel ID for a new channel, starting again from *segNr* = 1.

9.4 MSG_STAT

MSG_STAT uses *msgType* **0x02**, with *msgData* as follows:

byte	(batchNr << 3) | flags	*batchNr* communicates the batch number this MSG_STAT is in response to. See *Batch Number* in DTG_ENVELOPE. *flags* is a combination of flag bits. See *MSG_STAT Flags* below.
varuint	xmitNrLo	The lowest *xmitNr* successfully received by the sender of this MSG_STAT, in the batch this MSG_STAT is in response to, since sending the last MSG_STAT. Zero if nothing received since last MSG_STAT was sent.
varuint	xmitNrHi	The highest *xmitNr* received by the sender of this MSG_STAT, in the batch this MSG_STAT is in response to. Zero if nothing received yet. Set to same value as in last MSG_STAT if nothing received since last MSG_STAT.
nstime	xmitNrHiAge	The elapsed period of time, in nanoseconds, between when the datagram identified by *xmitNrHi* was received, and when this MSG_STAT was sent. Zero if nothing received yet.
varuint	nrReceived	The number of DTG_ENVELOPE successfully received by the sender for this batch since the sender's last MSG_STAT, not counting duplicates.
varuint	connParamsRev	The sender's *ConnParams* revision encoded in this MSG_STAT. Zero if no *ConnParams* is encoded. The sender may stop encoding *ConnParams* if it hasn't changed since the last acknowledgment received from the other party.
ConnParams	connParams	Present if *connParamsRev* is not zero. See *Connection Parameters* below.
varuint	connParamsAck	The highest *ConnParams* revision which the sender of this MSG_STAT has received from the receiver of the MSG_STAT. Zero if no *ConnParams* received yet.
vect<ChanStat>	channelStats	Statistics and missing segments for zero or more channels. See *Channel Statistics and Missing Segments* below.

9.4.1 Sending Frequency

In the absence of other connection activity, MSG_STAT is the heartbeat of the connection that continues when the connection is otherwise inactive. If no MSG_STAT is received in a reasonable time, an implementation can assume disconnection. See also *Sessions and Ports* > *Connection Duration*.

MSG_STAT is sent:

- By the server as soon as it sends DTG_KEXREPLY, and by the client as soon as it receives it.
- Immediately after a MSG_STAT with the STATFL_WANTREPLY flag is received. See *Flags* below.
- Immediately after a MSG_STAT is received with a new batch number.
- Not more than ⅔ of round trip time after receiving a MSG_DATAGRAM or MSG_CHANNELDATA. Can be sent more frequently to prevent MSG_STAT size from exceeding optimal datagram size.
- When no MSG_DATAGRAM or MSG_CHANNELDATA has been received since the last MSG_STAT, the time until the next MSG_STAT is doubled, and continues to be doubled each time MSG_STAT is sent, as long as no MSG_DATAGRAM or MSG_CHANNELDATA is received.
- Notwithstanding the above, the maximum time between two MSG_STAT is 5 seconds.

9.4.2 MSG_STAT Flags

The *flags* field in MSG_STAT is a bitwise-OR combination of bit values. Unassigned bits **must** be set to zero by the sender, and ignored by the receiver. Currently, one bit value is defined:

0x01	STATFL_WANTREPLY	The sender requests that the receiver send an immediate MSG_STAT, promptly after receiving the datagram in which this flag is set. The MSG_STAT that is sent in reply may also have this flag set, in which case it should also be honored. This can be used by implementations to measure roundtrip time in the absence of data to send.

9.4.3 Connection Parameters

The *ConnParams* type is encoded as follows:

`uint16`	`optInSize`	The maximum UDP datagram size the sender of this MSG_STAT permits the other party to send. A party must **not** send datagrams containing MSG_CHANNELDATA whose UDP datagram size exceeds the limit in the last acknowledged *ConnParams*, except when resending missing segments, which must retain the size that was originally sent. A receiver **must** accept MSG_CHANNELDATA datagrams of size up to the largest *optInSize* that the receiver sent in any *ConnParams* during the same connection. This value **must** be at least 500. Before the first *ConnParams* is received, both parties assume a default value of 500.
`varuint`	`maxInChannelId`	The highest channel ID that will be accepted by the sender of this MSG_STAT for channels opened by the receiver. When sending *ConnParams*, a party may only ever increase this number; if the number decreases, the receiver **must** ignore the new limit. A party must **not** open a channel with an ID exceeding the largest *maxInChannelId* received from the other party. Before the first *ConnParams* is received, both parties assume a default value of zero. This initially permits the client only to open one channel with ID zero.
`varuint`	`newBufferCount`	The number of segments the sender of this MSG_STAT is willing to buffer for a new channel opened by the other party. This is a guideline, not a guarantee – a party may dynamically adjust its buffering capacity at any time, resulting in segments being dropped that are below a recently advertised limit.

9.4.4 Channel Statistics and Missing Segments

The *ChanStat* type is encoded as follows:

varuint	channelId	See *channelId* in MSG_CHANNELDATA.
varuint	chanFlags	A combination of flag bits. See *ChanStat Flags* below. **The remaining fields below** are encoded only if **none** of the following ChanStat flags are present: CHANFL_RECVCLOSED, CHANFL_CLOSED, and CHANFL_CLOSEDACK. For the purpose of making this encoding/decoding decision, the values of any other flag bits, including those not yet defined, **must** be ignored.
varuint	hiSegNr	The highest *segNr* received by sender on this channel from the other party. Zero if no data received yet.
varuint	canBufferCount	The number of segments, counting from the first segment after *lastRelayed*, which the sender of this MSG_STAT is willing to buffer. Any received segments further along the stream will be dropped. This is a guideline, not a guarantee – a party may dynamically adjust its buffering capacity at any time, resulting in segments being dropped that are below an advertised limit.
varuint	totalNrMissing	The total number of missing segments. This number can be higher than the number of *segNr* enumerated by *missing*.
varuint	lastRelayed	The last *segNr* on this channel which the sender of this MSG_STAT has relayed to an upper layer. Zero if no segments have been accepted by an upper layer. The *canBufferCount* value is relative to this value.
varuint	baseMissing	The first missing *segNr* enumerated in this *ChanStat*. Zero if this *ChanStat* does not enumerate any missing segments.
vect<Missing>	missingRanges	Encodes a subset of *segNr* which the sender has not yet received. When the set of missing segments does not fit into one MSG_STAT, the sender may split it into multiple MSG_STAT, or may delay sending information about later missing segments until earlier ones are received.

The *Missing* type is encoded as follows:

varuint	nrMissing	The number of sequential missing segments in this span.
varuint	nrReceived	The number of sequential received segments after this span.

9.4.5 ChanStat Flags

The *chanFlags* field of *ChanStat* is a bitwise-OR combination of bit values. Unassigned bits **must** be set to zero by the sender, and ignored by the receiver. Currently, the following bit values are defined:

0x01	CHANFL_RECVCLOSED	The sender of MSG_STAT is ignoring any further data received on this channel. The receiver of MSG_STAT should discard any data it was still going to send, and not send any more segments. Note that this flag does **not** automatically imply that the channel is being closed; the other sending direction can remain open independently. An application **must** send CHANFL_RECVCLOSED after it has received an empty segment signaling End of Data, **and** has also successfully received **and** relayed all segments up to and including End of Data to the higher layer channel implementation.
0x02	CHANFL_CLOSED	The sender of MSG_STAT is indicating that it considers the channel closed. It will send no further data on the channel, it will discard incoming data on the channel, and has released channel resources other than those required to handle CHANFL_CLOSED. During normal channel closure, a party sends CHANFL_CLOSED when it is satisfied that it has received EOD from the other party, and that the other party has acknowledged EOD from this party. A party may also abandon a channel by cleaning up most channel resources and sending CHANFL_CLOSED without waiting for EOD signals. When a party sends CHANFL_CLOSED, it continues to include it in MSG_STAT datagrams until the other party responds with its own CHANFL_CLOSED for the same channel. After CHANFL_CLOSED is received, a party can completely free the channel when more than 60 seconds have elapsed after the last CHANFL_CLOSED was received. Neither party sets this flag after it has received CHANFL_CLOSEDACK.
0x04	CHANFL_CLOSEDACK	The sender of MSG_STAT is indicating that it received the other party's CHANFL_CLOSED for this channel. This flag is sent in response to every CHANFL_CLOSED. When a party needs to send this flag because the other party sent CHANFL_CLOSED, the timer for completely freeing the corresponding channel is reset to zero.

9.5 MSG_ABORTSESSION

MSG_ABORTSESSION uses *msgType* **0x03**, with *msgData* as follows:

`string`	`reason`	An optional UTF-8 string explaining the reason for session abort. If present, it can be logged or shown to an operator.

A graceful EST session shutdown does not involve MSG_ABORTSESSION, but is instead a natural result of timeout when there are no open channels. MSG_ABORTSESSION is sent by a party to signal that it's abandoning the connection and doesn't care about the cost. The cost is that any outstanding data not yet fully transmitted on any channel will be discarded.

An application that's in a hurry to close a connection, e.g. because it is shutting down in a way that provides minimal time for cleanup, **may** send a burst of identical MSG_ABORTSESSION datagrams before it exits, relying on that at least one of them is likely to arrive.

10 Congestion Avoidance

June 2019: Do not implement the congestion control algorithm proposed in this section. Better congestion control algorithms exist – perhaps BBR as presented by Google at IETF 97 in Seoul, 2016:

https://datatracker.ietf.org/meeting/97/materials/slides-97-iccrg-bbr-congestion-control

This proposal rested on the assumption that packet loss is going to increase transparently if sending rate exceeds bandwidth, but there exist actual networks that behave catastrophically in this case. In such networks, if sending rate exceeds bandwidth, the router tries to store all packets and make sure they get through until latency is so bad that *all* connections fail, not only the one that should be affected.

10.1 Overview

The EST protocol provides tools for applications to discover network conditions and adjust sending rates. EST implementations **must** determine sending rate either using the algorithm described herein, or an improvement that remains cooperative with and fair to implementations of this algorithm.

In designing this algorithm, I make the following assumptions:

- The network route between two communicating EST nodes has a maximum bandwidth, B, which can change from time to time during a session, depending on how traffic is routed. If total traffic volume V exceeds B, datagrams will be dropped randomly with probability $\frac{B}{V}$.
- If the network route is unreliable, it will lose datagrams randomly and at a small percentage of overall traffic. This algorithm's tolerance for random loss grows smaller as bandwidth increases.
- An EST connection may compete with a number of other agents sending traffic using the same network route. The algorithm described herein assumes such agents will reduce their sending rate when they experience packet loss (i.e. when total traffic volume exceeds B).

10.1.1 Explicit Congestion Notification

EST can be extended to support ECN in the future. This might be implemented as follows:

- The parties signal ECN support through an extension in DTG_INIT and DTG_INITREPLY.
- When both parties support the ECN extension, the *ChanStat* structure in MSG_STAT is extended with (1) a bit for the receiver to signal that valid MSG_CHANNELDATA were received for that channel with Congestion Encountered bits set in the IP header, and (2) a bit for the sender to signal that the sending rate was reduced.

The main reasons this document does not specify ECN support at this time are:

- OS networking APIs do not make ECN bits accessible to applications using UDP sockets.
- Support for ECN in deployed routers is sketchy – many networks will clear ECN bits, others will outright drop IP packets with ECN bits set.

- Depending on the characteristics of a router implementation, an application that implements ECN may suffer from surrendering bandwidth to other applications that wait for datagram loss before adjusting their sending bandwidth. This may make ECN-supporting applications appear lower-performing and unattractive, even though they're actually better behaved.

10.1.2 Congestion Avoidance Trigger

The congestion avoidance trigger for this algorithm is datagram loss, rather than detection of increasing roundtrip time, for the following reasons:

- Datagram loss is easily measurable and unambiguous. Congestion avoidance based on roundtrip time requires a correct measurement of base roundtrip time, which may not be available.
- An algorithm that avoids congestion before it happens is surrendering bandwidth for the benefit of algorithms that wait for actual datagram loss to happen. In the current internet, algorithms that use datagram loss as trigger are predominant.
- Waiting for datagram loss does mean that, if the network bottleneck is at a router with overly large buffers, bufferbloat will occur. However, I believe this is a problem to be fixed by router operators, not protocols. If a router cannot relay a packet without significant delay, it should drop it. If the router doesn't do that, the resulting bufferbloat is the router's fault.

10.2 Algorithm

10.2.1 Sending Rate – V

This algorithm uses the concept of sending rate, V, which is expressed as the number of bytes that an application will allow itself to send per second. Each EST connection has two separate sending rates, one maintained by each party for its sending direction. There is no maximum sending rate. The minimum value is 1,000 bytes per second.

10.2.2 Roundtrip Time – R

This algorithm uses the concept of roundtrip time, R, similar to that used in TCP. Each party maintains a single R value per session by observing the delay between when datagrams are sent, and when acknowledgment of their receipt arrives through *xmitNrHi* in MSG_STAT.

10.2.3 Loss Event Threshold – L

EST implements a loss event threshold that is always more forgiving than the threshold used by TCP, but grows increasingly less forgiving depending on the current sending rate. The loss event threshold is calculated from the sending rate, V, as follows:

$$L = \frac{10}{\sqrt{V}}$$

The result, L, is the proportion of datagrams that need to be reported lost for a loss event to be triggered. The above formula produces the following values in practice:

Bytes/second – V	*Loss threshold – L*	*Interpretation – assumes datagrams of 1,000 bytes*
1 000	0.31623	Can't lose datagrams without scaling back
10 000	0.10000	Can lose 1 datagram/s without scaling back
100 000	0.03162	Can lose 3 datagrams/s without scaling back
1 000 000	0.01000	Can lose 10 datagrams/s without scaling back
10 000 000	0.00316	Can lose 31 datagrams/s without scaling back
100 000 000	0.00100	Can lose 100 datagrams/s without scaling back
1 000 000 000	0.00032	Can lose 316 datagrams/s without scaling back
10 000 000 000	0.00010	Can lose 1,000 datagrams/s without scaling back

This approach has the following properties:

- Much more resilient to random data loss than the usual TCP threshold. The slower the sending rate, the more resilience we have to random loss due to an unreliable (e.g. wireless) connection.
- When multiple EST data flows share the same network path, faster data flows will tend to detect and react to congestion faster than slower flows, making it easier for slower flows to catch up.

10.2.4 Scale-Back Time Offset – f

EST scales back its sending rate by shifting the time parameter used to calculate window size, rather than by manipulating the window size directly. The scale-back time offset, f, is calculated from the sending rate, V, as follows:

$$f = \frac{6}{3 + \log_{10} V}$$

The above formula produces the following values in practice:

Bytes/second – V	*Scale-back time offset – f*	*V / V_{max} after scale-back*
1 000	1.000	0.963
10 000	0.857	0.921
100 000	0.750	0.875
1 000 000	0.667	0.829
10 000 000	0.600	0.784
100 000 000	0.545	0.742
1 000 000 000	0.500	0.704
10 000 000 000	0.462	0.668

This approach has the property that faster data flows will scale back more than slower flows, further making it easier for slower flows to catch up.

10.2.4 Discovery Phase

An implementation **may** start congestion avoidance in *Maintenance Phase* if it has access to recent estimates of R and V_{max} that it believes will be valid for the current connection. Otherwise, at the start of an EST connection, the congestion avoidance algorithm starts in *Discovery Phase*.

When starting or restarting *Discovery Phase*, the application resets $R = 1s$, $V = 32\,768\ bytes/s$, and presumes to know no value for V_{max}. During this phase, the application increases V by a factor of two for every roundtrip time when no loss event is triggered, but MSG_STAT messages are being received.

When a loss event occurs, the following is done:

- V_{max} is set to equal the last sending rate, V, multiplied by the percentage of datagrams that were being successfully delivered when the loss event was triggered.
- The scale-back time offset f is calculated based on V_{max}.
- The application proceeds to *Maintenance Phase*.

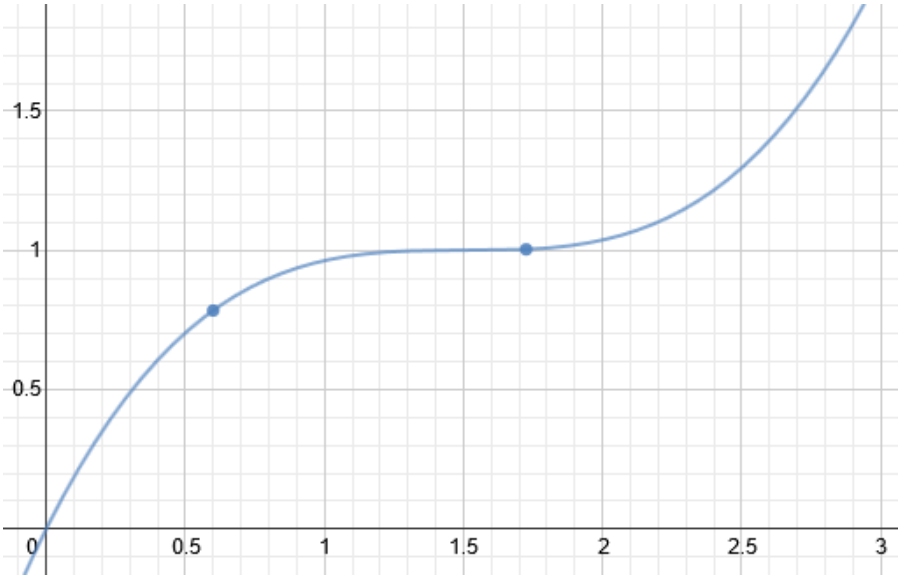
If the application enters a period of inactivity and then resumes sending, the sending rate is reset to equal $V = V \cdot 0.975^{(t-t_l)}$, where t_l is the time of the last send. This reverts window progress to a size below what it was for the last send, progressively lower the longer the period of inactivity.

10.2.5 Maintenance Phase

In *Maintenance Phase*, the application has established values for R, V, V_{max} , and f. In this phase, the application measures the time t (in seconds) since the last loss event was triggered, and continuously updates the sending rate as follows:

$$V = \left(\frac{2}{3}(f + t) - 1\right)^3 + V_{max}$$

This is a cubic function which looks as follows:



The X axis above represents $f + t$ instead of t; the function begins to be evaluated around the left dot, when $t = 0$, but $f + t > 0$. The Y axis is defined so that $V_{max} = 1$.

The properties of this function are as follows:

- When $t = 0$, starts with V as a percentage of V_{max} – see table in *Scale-Back Time Offset*.
- Grows faster at first, and then slowly towards $V = V_{max}$, meeting V_{max} when $f + t = 1.5$.
- After passing V_{max}, grows slowly at first, then increasingly faster if no loss event occurs.

When a loss event occurs, the following is done:

- V_{max} is reset to equal the last sending rate, V, multiplied by the percentage of datagrams that were being successfully delivered when the loss event was triggered.
- The scale-back time offset f is calculated based on V_{max}.
- The time parameter is reset to $t = 0$.

When network conditions are stable, a loss event is expected to occur around the right dot on the illustrated curve, causing a restart from the left dot (the scale-back origin point). Depending on V, which dictates the exact values of f and L, the cycle will have a period of around 1 second.

If the application enters a period of inactivity and then resumes sending, the following is done:

- The scale-back time offset is set to $f = (f + t_l) \cdot 0.975^{(t - t_l)}$, where t_l is the time of last send.
- The time parameter is reset to $t = 0$. Through these updates, sending rate progress is reverted to a state before the last send, progressively earlier the longer the period of inactivity.
- V_{max} is unchanged.

10.2.6 Timeout

If a timeout occurs during either *Discovery Phase* or *Maintenance Phase* – i.e. no MSG_STAT is received from the receiver, and there are sent MSG_CHANNELDATA outstanding – then all MSG_CHANNELDATA that were sent but not yet reported as received are marked for resending, and *Discovery Phase* is restarted with all parameters reset to initial values.

11 Cryptography

11.1 Discussion

11.1.1 Algorithm Variety

The main objective of algorithm negotiation in a protocol like EST is to enable a smooth transition when an impending weakness is discovered in a previously favored algorithm, but that algorithm remains useful (although weaker) for a time.

Another important objective is to allow implementations to experiment with and deploy algorithms and extensions which might be of interest to some users, but unforeseen during protocol design.

A final objective is to provide a defense against a situation where a favored, previously trusted algorithm is thoroughly broken in a short time, so that anyone can exploit it at low cost. This is an unlikely and rare event that is hard to plan for. I expect that protection against this unlikely scenario will evolve incrementally, as new algorithms are specified over time. If this scenario arises with a public key algorithm, the automatic fingerprint distribution and revocation features of EST make it more suited for the challenge than SSH.

11.1.2 Algorithm Choice

At this time, the *secp256k1* curve is thought to provide the equivalent of 128-bit symmetric security. The choice of 128-bit AES for encryption is consistent with this choice, is significantly faster than 256-bit AES, and does not have key scheduling issues in 256-bit AES. The hash being used is SHA2-256 due to progress in SHA1 cryptanalysis, and due to SHA3 not yet being standardized.

This document specifies ECC-based public key cryptography due to a perceived reasonable chance that breakthroughs in RSA and classic DH cryptanalysis may render them insecure in foreseeable future.

11.1.3 Patent Encumbrance

I believe that ECDH and ECDSA as described herein can be implemented without patent encumbrance using basic, public domain cryptographic techniques, e.g. as implemented in LibTomCrypt, for one example.

This specification does require implementations to support ECC point compression, which is claimed in U.S. Patent 6,141,420. This patent expires July 29, 2014, and will therefore be irrelevant by the time most implementors consider EST.[3]

[3] According to Daniel J. Bernstein, it may have always been irrelevant, and would not survive a court challenge. See his discussion of this claim here: http://cr.yp.to/patents/us/6141420.html

11.1.4 Random Number Generation

EST implementations **must** ensure that **all** random numbers generated for use in the protocol are cryptographically random. The following is a non-exhaustive list of security suggestions:

- Do not trust the operating system to provide an infallible source of randomness. Some OS RNG implementations are faulty. Others might be backdoored. Most are vulnerable to restoring virtual machine state, which can lead directly to private key compromise when using non-deterministic DSA or ECDSA signatures.
- If at all possible, use a random pool with entropy continuously replenished from multiple sources: the RNG provided by the OS, the application's own activity, the data the application is handling. Do not overestimate the amount of entropy obtained from each individual source – many processes that at first seem unpredictable, are in fact surprisingly predictable, given even a moderate amount of study.
- If at all possible, take explicit steps to replenish random pool entropy with the current timestamp and the contents of the current message immediately before performing a sensitive operation such as an ECDSA signature, or generating xmitNonce for DTG_ENVELOPE.
- Avoid known backdoored algorithms, such as Dual_EC_DRBG.

11.1.5 Conventions

Algorithm definitions in this section use the type *byteString* to describe a raw array of bytes internally represented in an application-specific manner. This term is used instead of previously defined *string* to indicate that when an instance of *byteString* is processed, the bytes of the string are processed as they appear, without a leading *uint16* to indicate length.

11.2 EST-HASH

EST attempts to let parties negotiate their algorithm choice as much as possible, but there are contexts where a standardized hashing algorithm is required, such as for public key fingerprint verification, and in datagrams exchanged during algorithm negotiation itself.

For use in these contexts, the following hash function is defined:

```
EST-HASH(X) = first 20 bytes of HMAC-SHA2-256(K: ”est-hash”, m: X)
```

Since any algorithm chosen to serve as EST-HASH will be hard to replace without causing interoperability issues, it is the hope of this specification that the above construct will remain sufficiently secure for a long time, even in the event that modest weaknesses are discovered in the underlying hash function.

11.3 Key Exchange

The abstract interface of a generic EST key exchange algorithm is:

```
ClientInitiate – function:
    output: clientEphemeralPrivateKey – byteString
            msgForServer              – byteString

ServerReply – function:
    input:  msgForServer              – byteString
    output: valid                     – boolean
            agreedValue               – byteString
            msgForClient              – byteString

ClientVerify – function:
    input:  clientEphemeralPrivateKey – byteString
    output: valid                     – boolean
            agreedValue               – byteString
```

11.3.1 Algorithm "ecdh"

All implementations are **required** to support this algorithm, but users can disable it in implementations that support other key exchange algorithms that may be defined in the future.

The algorithm used for shared key generation is ECDH with incompatible cofactor multiplication, as specified in SEC 1, section 3.3.2.

Ephemeral keypairs are generated as specified in SEC 1, section 3.2.1.

Implementations **must** validate ephemeral public keys received from the other party. A validation algorithm can be found in SEC 1, section 3.2.2. If a key fails validation, the key exchange packet must be discarded as invalid. An implementation **should** retain handshake state for a time, in the event that a valid packet arrives, since the invalid one could have been spoofed.

The ephemeral public keys are elliptic curve points, and must be encoded for transmission. This is done according to encoding rules described in SEC 1, sections 2.3.3 and 2.3.4. Implementations **may** use point compression, and **must** accept compressed curve points.

The shared secret produced by ECDH over *secp256k1* is a curve point, which is trivially converted into an integer as per SEC 1, section 2.3.9. To produce *agreedValue*, this integer is encoded as a fixed-size array of 32 bytes, **most** significant byte first[4]. Note that this is subtly different from key exchange in SSH, where the agreed value is encoded as a variable-length signed integer.

For this algorithm, *msgForServer* is encoded as follows:

`byte[…]`	`clientEphemeralPublicKey`	The public key component of the client's ephemeral ECDH keypair. No size is encoded because it is implied in the size of *msgForServer*.

Similarly, *msgForClient* is encoded as follows:

`byte[…]`	`serverEphemeralPublicKey`	The public key component of the server's ephemeral ECDH keypair. No size is encoded because it is implied in the size of *msgForClient*.

[4] This is what Crypto++ ECDH Agree produces.

11.4 Public Keys

The abstract interface of a generic EST public key algorithm is:

```
Generate – function:
    output: privateKey – byteString
            publicKey  – byteString

Sign – function:
    input:  privateKey – byteString
            data       – byteString
    output: signature  – byteString

Verify – function:
    input:  publicKey  – byteString
            data       – byteString
            signature  – byteString
    output: valid      – boolean
```

11.4.1 Textual Public Key Format

Applications that participate in exchange of public keys outside of the EST protocol **must** support a textual public key file format where each non-empty line contains a public key encoded as follows:

```
optHws keyword hws pubKeyBase64 hws comment
```

Fields of the textual public key format are defined as follows:

optHws	Optional horizontal whitespace (any combination of TAB and SP characters).
keyword	The string “pub” (without quotes).
hws	Non-optional horizontal whitespace (at least one TAB or SP character).
pubKeyBase64	A Base64 encoding (RFC 4648) of the binary public key blob as defined in *Generic Public Key Format*.
comment	An optional description of the public key, terminated by end of line. The comment **must** use UTF-8 encoding.

When stored automatically by an application, a text file in the above format **must** be stored with a name that begins with the string ”public”, and ends in the extension “.txt” (both case-insensitive). The filename **may** contain zero or more other characters between “public” and “.txt”. This filename pattern is **required** to reduce the chance that users will compromise themselves by sharing a private key instead of a public key. The prefix is important for visibility of the file type on systems which hide file extensions by default. For example, the following are acceptable filenames:

```
public.txt
public0.txt
publicKeys-SuperApp.txt
```

11.4.2 Binary Private Key Format

Private keys are not exchanged within EST as currently defined, but implementors **should** support this common private key format to transfer private keys to and from other EST applications.

To reflect the encrypted and sensitive nature of private keys, applications **should** store and exchange them in binary files. Use of a textual format for private keys is **discouraged** to distinguish them from public keys (which may be kept in a textual format), and so that users might be less likely to send someone a private key that they should keep secret.

A group of private keys are encoded in a *MasterKeys* type, as follows:

`name`	`encAlg`	The name of the encryption algorithm, or empty if none. If non-empty, the fields that follow are encrypted.
`vect<MasterKey>`	`masterKeys`	A series of *MasterKey* type entries.
`vect<esthash>`	`revokedFps`	Fingerprints of keys that may have been used previously, and have now been revoked.

An individual *MasterKey* is encoded as follows:

`PubKey`	`pubKey`	The master public key, as defined in *Generic Public Key Format*.
`string`	`comment`	A user-provided or automatically generated description of the keypair. **Must** use UTF-8 encoding. May be empty.
`string`	`privKey`	The master private key, encoded in an algorithm-specific manner.
`varuint`	`sigCount`	The number of signatures this master key has performed.
`vect<SubKey>`	`subKeys`	The master key's current sub-certificates and corresponding private keys, encoded as a series of *SubKey* types.

An individual *SubKey* is encoded as follows:

`SubCert`	`subCert`	The public sub-certificate, encoded as defined in *Sub-Certificates*.
`string`	`privKey`	The private key associated with the sub-certificate, encoded in an algorithm-specific manner.
`varuint`	`sigCount`	The number of signatures this sub-certificate has performed.

To avoid performing disk I/O for every signing operation, an application **may** store fewer than the actual number of sub-certificates. When an application loads a private key with fewer sub-certificates than called for by the public key algorithm, it **must** add newly generated sub-certificates up to the algorithm-specified sub-certificate chain depth.

When stored automatically by an application, a private key file **must** be stored with a name that begins with the string "private", and ends in the extension ".dat" (both case-insensitive). The filename **may** contain zero or more other characters between "private" and ".dat". This filename pattern is **required** to reduce the chance that users will compromise themselves by sharing a private key instead of a public key. The prefix is important for visibility of the file type on systems which hide file extensions by default. For example, the following are acceptable filenames:

```
private.dat
private0.dat
privateKeys-SuperApp.dat
```

11.4.3 Private Key Encryption Algorithm "aes-pbkdf2"

This private key encryption algorithm takes as input:

1. A string of bytes representing a user-entered password in UTF-8 encoding.
2. An iteration count which should be configurable by the user, and **should** permit large values on the order of millions.
3. The private key blob produced by the public key algorithm. This is *not* the blob defined in *Binary Private Key Format*, but rather the algorithm-specific key blob that would be stored in *privateKeyData* if it was stored unencrypted.

This encryption algorithm is defined with pseudo-code as follows:

```
let PBKDF2 be the PBKDF2 function from PKCS #5 (RFC 2898) using SHA2-256

let password = user-entered password in UTF-8
let iterations = iteration count
let plaintext = encoded private keys and revoked fingerprints to encrypt

let salt = a byteString consisting of 8 or more randomly generated bytes
let keyBase = PBKDF2(password, salt, iterations, 32)
let encKey = truncate(SHA2-256(SHA2-256(concatenate(keyBase, "enc")), 16)
let macKey = truncate(SHA2-256(SHA2-256(concatenate(keyBase, "mac")), 16)
let iv  = truncate(SHA2-256(SHA2-256(concatenate(keyBase, "iv" )), 16)

let saltS = encode salt as string
let iterationsS = encode iterations as varuint
let ciphertext = AES128-CTR(encKey, iv, plaintext)
let macData = concatenate(saltS, iterationsS, ciphertext)
let mac = HMAC-SHA2-256(macKey, macData)
output concatenate(macData, mac)
```

11.4.4 Algorithm "ecdsa"

All implementations are **required** to support this algorithm, but users can disable it in implementations that support other public key algorithms that may be defined in the future.

The signing and verification algorithm used is ECDSA as specified in SEC 1, section 4.1, using SHA2-256 as the hash function, and *secp256k1* as the elliptic curve.

A keypair is generated as specified in SEC 1, section 3.2.1.

Public keys are elliptic curve points, and must be encoded for transmission. This is done according to encoding rules described in SEC 1, sections 2.3.3 and 2.3.4. Implementations **may** use point compression, and **must** accept compressed curve points.

A private key is encoded as a fixed-size 32-byte integer, **most** significant byte first.

A signature is encoded as a concatenation of the two ECDSA result values, first *r*, then *s*, both represented as fixed-size 32-byte integers, **most** significant byte first[5].

Implementations **must** ensure that the signing parameter *k* is generated in a way that defends against situations where the same *k* might be used for more than one signature. One way to do this is to generate *k* deterministically depending on the private key and message, as discussed in RFC 6979. Another way is to feed the entire message as entropy into an RNG that can incorporate entropy, before using it to generate *k*.

The sub-certificate chain depth used by this algorithm is a total of 7 sub-certificates. The maximum number of signatures performed by each sub-certificate before replacing it is 100. These numbers are chosen for the following reasons:

- Although properly implemented ECDSA is not known to leak private key data through signature content, implementations are highly likely to leak private data through side channels while signing.
- ECDSA key generation is fast compared to signing.
- 100^8 keys are sufficient to generate 10 million signatures per second, for 31 years, before running out of keys. Therefore, applications do not need to throttle signing.

[5] This is what the Crypto++ ECDSA Verifier consumes, and Signer produces.

11.5 Encryption and Integrity

11.5.1 Re-Keying

An EST encryption and integrity algorithm **must** be careful not to encrypt more data using the same cryptographic parameters as can be securely encrypted by the primitives it uses. A good way to do so is to generate new encryption keys every time *xmitNr* passes a particular value; this is done by the algorithm defined below.

11.5.2 Abstract Interface

The abstract interface of a generic EST encryption and integrity algorithm is:

```
BlockSize – integer constant, algorithm-specific, in bytes

EncryptedLen – function:
    input:  plaintextLen  – integer
    output: ciphertextLen – integer

Encrypt – function:
    input:  sharedSecret  – byteString
            xmitNr        – integer
            envelope      – byteString, the fields of DTG_ENVELOPE that
                            precede encrypted data
            plaintext     – byteString, length must be a multiple of BlockSize
    output: ciphertextMac – byteString,
                            length can be more or less or equal to plaintext

Decrypt – function:
    input:  sharedSecret  – byteString
            xmitNr        – integer
            envelope      – byteString, the fields of DTG_ENVELOPE
                            that precede encrypted data
            ciphertextMac – byteString, length must be a multiple of BlockSize
    output: plaintext     – byteString, length can be more or less or
                            equal to plaintext
```

11.5.3 Algorithm "aes-ctr"

All implementations are **required** to support this algorithm.

This algorithm is defined with pseudo-code as follows:

```
BlockSize: 1 (CTR mode does not require padding)

EncryptedLen:
    output plaintextLen + 16

Encrypt:
    let DoubleSha(X) = SHA2-256(SHA2-256(X))

    let direction = "ToServer" if sending to server, "ToClient" otherwise
    let keyLabel  = concatenate("EncKey", direction)
    let macLabel  = concatenate("MacKey", direction)
    let ivLabel   = concatenate("Iv",  direction)
    let keyBase   = DoubleSha(concatenate(DoubleSha(sharedSecret), keyLabel))
    let macBase   = DoubleSha(concatenate(DoubleSha(sharedSecret), macLabel))
    let ivBase    = DoubleSha(concatenate(DoubleSha(sharedSecret), ivLabel))
    ' keyBase, macBase, and ivBase remain constant the entire session.

    let keyCycle = xmitNr / 1,000 (integer division)
    let keyCycleS = encode keyIndex as 64 bits, most significant byte first
    let encKey = truncate(DoubleSha(concatenate(keyBase, keyCycleS)), 16)
    let macKey = truncate(DoubleSha(concatenate(macBase, keyCycleS)), 16)
    let ivSeed = DoubleSha(concatenate(ivBase, keyCycleS))
    ' encKey, macKey and ivSeed change once every 1,000 datagrams

    let xmitNrS = encode xmitNr as 64 bits, most significant byte first
    let iv = truncate(SHA2-256(concatenate(ivSeed, xmitNrS, xmitNonce)), 16)

    let ciphertext = AES128-CTR(encKey, iv, plaintext)
    let mac = HMAC-SHA2-256(macKey, concatenate(envelope, ciphertext))
    output ciphertextMac = concatenate(ciphertext, truncate(mac, 16))

Decrypt:
    Derive keys as in Encrypt, then authenticate using HMAC-SHA2-256,
    and decrypt using 128-bit AES in CTR mode.
```

11.5.4 Algorithm "aes-gcm"

Implementations are **recommended** to support this algorithm, and to favor it over "aes-ctr". Support for this algorithm is not *required* because GCM mode may not be easily available to all implementations, e.g. those using FIPS-certified modules which do not yet support GCM.

This algorithm is defined with pseudo-code as follows:

```
BlockSize: 1 (GCM mode does not require padding)

EncryptedLen:
    output plaintextLen + 16

Encrypt:
    let DoubleSha(X) = SHA2-256(SHA2-256(X))

    let direction = "ToServer" if sending to server, "ToClient" otherwise
    let keyLabel  = concatenate("EncKey", direction)
    let ivLabel   = concatenate("Iv",  direction)
    let keyBase   = DoubleSha(concatenate(DoubleSha(sharedSecret), keyLabel))
    let ivBase    = DoubleSha(concatenate(DoubleSha(sharedSecret), ivLabel))
    ' keyBase and ivBase remain constant the entire session.

    let keyCycle = xmitNr / 1,000 (integer division)
    let keyCycleS = encode keyIndex as 64 bits, most significant byte first
    let key = truncate(DoubleSha(concatenate(keyBase, keyCycleS)), 16)
    let ivSeed = DoubleSha(concatenate(ivBase, keyCycleS))
    ' key and ivSeed change once every 1,000 datagrams

    let xmitNrS = encode xmitNr as 64 bits, most significant byte first
    let iv = truncate(SHA2-256(concatenate(ivSeed, xmitNrS, xmitNonce)), 16)

    let AES128-GCM be 128-bit AES encryption in Galois Counter Mode,
        taking as input: data to authenticate, key, iv, data to encrypt,
        and producing as output a concatenation of ciphertext and 128-bit tag

    output ciphertextMac = AES128-GCM(envelope, key, iv, plaintext)

Decrypt:
    Derive keys as in Encrypt, then decrypt and authenticate using
    128-bit AES in GCM mode.
```

12 Resources

SEC 1	Standards for Efficient Cryptography Group: ***Elliptic Curve Cryptography*** http://www.secg.org/download/aid-780/sec1-v2.pdf
SEC 2	Standards for Efficient Cryptography Group: ***Recommended Elliptic Curve Domain Parameters*** http://www.secg.org/download/aid-386/sec2_final.pdf
RFC 6979	T. Pornin: ***Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA)*** http://tools.ietf.org/html/rfc6979 See also errata: http://www.rfc-editor.org/errata_search.php?rfc=6979