Tor Protocol Specification

Roger Dingledine
Nick Mathewson

Note: This document aims to specify Tor as currently implemented, though it may take it a little time to become fully up to date. Future versions of Tor may implement improved protocols, and compatibility is not guaranteed. Compatibility notes are given for versions 0.1.1.15-rc and later. We may or may not remove compatibility notes for other obsolete versions of Tor as they become obsolete.

This specification is not a design document; most design criteria are not examined. For more information on why Tor acts as it does, see tor-design.pdf.

0. Preliminaries

> The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119.

0.1. Notation and encoding

PK -- a public key.
SK -- a private key.
K -- a key for a symmetric cipher.

a|b -- concatenation of 'a' and 'b'.

[A0 B1 C2] -- a three-byte sequence, containing the bytes with hexadecimal values A0, B1, and C2, in that order.

H(m) -- a cryptographic hash of m.

We use "byte" and "octet" interchangeably. Possibly we shouldn't.

0.1.1. Encoding integers

Unless we explicitly say otherwise below, all numeric values in the Tor protocol are encoded in network (big-endian) order. So a "32-bit integer" means a big-endian 32-bit integer; a "2-byte" integer means a big-endian 16-bit integer, and so forth.

0.2. Security parameters

Tor uses a stream cipher, a public-key cipher, the Diffie-Hellman protocol, and a hash function.

KEY_LEN -- the length of the stream cipher's key, in bytes.

PK_ENC_LEN -- the length of a public-key encrypted message, in bytes.
PK_PAD_LEN -- the number of bytes added in padding for public-key encryption, in bytes. (The largest number of bytes that can be encrypted in a single public-key operation is therefore PK_ENC_LEN-PK_PAD_LEN.)

DH_LEN -- the number of bytes used to represent a member of the Diffie-Hellman group.
DH_SEC_LEN -- the number of bytes used in a Diffie-Hellman private key (x).

HASH_LEN -- the length of the hash function's output, in bytes.

PAYLOAD_LEN -- The longest allowable cell payload, in bytes. (509)

CELL_LEN(v) -- The length of a Tor cell, in bytes, for link protocol version v.

```
CELL_LEN(v) = 512    if v is less than 4;
            = 514    otherwise.
```

0.3. Ciphers

For a stream cipher, we use 128-bit AES in counter mode, with an IV of all 0 bytes.

For a public-key cipher, we use RSA with 1024-bit keys and a fixed exponent of 65537. We use OAEP-MGF1 padding, with SHA-1 as its digest function. We leave the optional "Label" parameter unset. (For OAEP padding, see ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-1/pkcs-1v2-1.pdf)

For the "ntor" handshake, we also use the Curve25519 elliptic curve group.

For Diffie-Hellman, we use a generator (g) of 2. For the modulus (p), we use the 1024-bit safe prime from rfc2409 section 6.2 whose hex representation is:

```
"FFFFFFFFFFFFFFFFC90FDAA22168C234C4C6628B80DC1CD129024E08"
"8A67CC74020BBEA63B139B22514A08798E3404DDEF9519B3CD3A431B"
"302B0A6DF25F14374FE1356D6D51C245E485B576625E7EC6F44C42E9"
"A637ED6B0BFF5CB6F406B7EDEE386BFB5A899FA5AE9F24117C4B1FE6"
"49286651ECE65381FFFFFFFFFFFFFFFF"
```

As an optimization, implementations SHOULD choose DH private keys (x) of 320 bits. Implementations that do this MUST never use any DH key more than once.
[May other implementations reuse their DH keys?? -RD]
[Probably not. Conceivably, you could get away with changing DH keys once per second, but there are too many oddball attacks for me to be comfortable that this is safe. -NM]

For a hash function, we use SHA-1.

KEY_LEN=16.
DH_LEN=128; DH_SEC_LEN=40.
PK_ENC_LEN=128; PK_PAD_LEN=42

PK_ENC_LEN=128; PK_PAD_LEN=42.
HASH_LEN=20.

When we refer to "the hash of a public key", we mean the SHA-1 hash of the DER encoding of an ASN.1 RSA public key (as specified in PKCS.1).

All "random" values MUST be generated with a cryptographically strong pseudorandom number generator seeded from a strong entropy source, unless otherwise noted.

The "hybrid encryption" of a byte sequence M with a public key PK is computed as follows:
1. If M is less than PK_ENC_LEN-PK_PAD_LEN, pad and encrypt M with PK.
2. Otherwise, generate a KEY_LEN byte random key K.
   Let M1 = the first PK_ENC_LEN-PK_PAD_LEN-KEY_LEN bytes of M,
   and let M2 = the rest of M.
   Pad and encrypt K|M1 with PK. Encrypt M2 with our stream cipher,
   using the key K. Concatenate these encrypted values.

[XXX Note that this "hybrid encryption" approach does not prevent an attacker from adding or removing bytes to the end of M. It also allows attackers to modify the bytes not covered by the OAEP -- see Goldberg's PET2006 paper for details. We will add a MAC to this scheme one day. -RD]

1. System overview

Tor is a distributed overlay network designed to anonymize low-latency TCP-based applications such as web browsing, secure shell, and instant messaging. Clients choose a path through the network and build a ``circuit'', in which each node (or ``onion router'' or ``OR'') in the path knows its predecessor and successor, but no other nodes in the circuit. Traffic flowing down the circuit is sent in fixed-size ``cells'', which are unwrapped by a symmetric key at each node (like the layers of an onion) and relayed downstream.

1.1. Keys and names

Every Tor relay has multiple public/private keypairs:

These are 1024-bit RSA keys:

- A long-term signing-only "Identity key" used to sign documents and certificates, and used to establish relay identity.
- A medium-term TAP "Onion key" used to decrypt onion skins when accepting circuit extend attempts. (See 5.1.) Old keys MUST be accepted for at least one week after they are no longer advertised. Because of this, relays MUST retain old keys for a while after they're rotated.
- A short-term "Connection key" used to negotiate TLS connections. Tor implementations MAY rotate this key as often as they like, and SHOULD rotate this key at least once a day.

This is Curve25519 key:

- A medium-term ntor "Onion key" used to handle onion key handshakes when accepting incoming circuit extend requests. As with TAP onion keys, old ntor keys MUST be accepted for at least one week after they are no longer advertised. Because of this, relays MUST retain old keys for a while after they're rotated.

These are Ed25519 keys:

- A long-term "master identity" key. This key never changes; it is used only to sign the "signing" key below. It may be kept offline.
- A medium-term "signing" key. This key is signed by the master identity key, and must be kept online. A new one should be generated periodically.
- A short-term "link authentication" key. Not yet used.

The RSA identity key and Ed25519 master identity key together identify a router uniquely. Once a router has used an Ed25519 master identity key together with a given RSA identity key, neither of those keys may ever be used with a different key.

2. Connections

Connections between two Tor relays, or between a client and a relay, use TLS/SSLv3 for link authentication and encryption. All implementations MUST support the SSLv3 ciphersuite "TLS_DHE_RSA_WITH_AES_128_CBC_SHA" if it is available. They SHOULD support better ciphersuites if available.

There are three ways to perform TLS handshakes with a Tor server. In the first way, "certificates-up-front", both the initiator and responder send a two-certificate chain as part of their initial handshake. (This is supported in all Tor versions.) In the second way, "renegotiation", the responder provides a single certificate, and the initiator immediately performs a TLS renegotiation. (This is supported in Tor 0.2.0.21 and later.) And in the third way, "in-protocol", the initial TLS renegotiation completes, and the parties bootstrap themselves to mutual authentication via use of the Tor protocol without further TLS handshaking. (This is supported in 0.2.3.6-alpha and later.)

Each of these options provides a way for the parties to learn it is available: a client does not need to know the version of the Tor server in order to connect to it properly.

In "certificates up-front" (a.k.a "the v1 handshake"), the connection initiator always sends a two-certificate chain, consisting of an X.509 certificate using a short-term connection public key and a second, self-signed X.509 certificate containing its identity key. The other party sends a similar certificate chain. The initiator's ClientHello MUST NOT include any ciphersuites other than:
  TLS_DHE_RSA_WITH_AES_256_CBC_SHA

TLS_DHE_RSA_WITH_AES_128_CBC_SHA
SSL_DHE_RSA_WITH_3DES_EDE_CBC_SHA

In "renegotiation" (a.k.a. "the v2 handshake"),
the connection initiator sends no certificates, and
the responder sends a single connection certificate. Once the TLS
handshake is complete, the initiator renegotiates the handshake, with each
party sending a two-certificate chain as in "certificates up-front".
The initiator's ClientHello MUST include at least one ciphersuite not in
the list above -- that's how the initiator indicates that it can
handle this handshake. For other considerations on the initiator's
ClientHello, see section 2.1 below.

In "in-protocol" (a.k.a. "the v3 handshake"), the initiator sends no
certificates, and the
responder sends a single connection certificate. The choice of
ciphersuites must be as in a "renegotiation" handshake. There are
additionally a set of constraints on the connection certificate,
which the initiator can use to learn that the in-protocol handshake
is in use. Specifically, at least one of these properties must be
true of the certificate:
* The certificate is self-signed
* Some component other than "commonName" is set in the subject or
issuer DN of the certificate.
* The commonName of the subject or issuer of the certificate ends
with a suffix other than ".net".
* The certificate's public key modulus is longer than 1024 bits.
The initiator then sends a VERSIONS cell to the responder, which then
replies with a VERSIONS cell; they have then negotiated a Tor
protocol version. Assuming that the version they negotiate is 3 or higher
(the only ones specified for use with this handshake right now), the
responder sends a CERTS cell, an AUTH_CHALLENGE cell, and a NETINFO
cell to the initiator, which may send either CERTS, AUTHENTICATE,
NETINFO if it wants to authenticate, or just NETINFO if it does not.

For backward compatibility between later handshakes and "certificates
up-front", the ClientHello of an initiator that supports a later
handshake MUST include at least one ciphersuite other than those listed
above. The connection responder examines the initiator's ciphersuite list
to see whether it includes any ciphers other than those included in the
list above. If extra ciphers are included, the responder proceeds as in
"renegotiation" and "in-protocol": it sends a single certificate and
does not request
client certificates. Otherwise (in the case that no extra ciphersuites
are included in the ClientHello) the responder proceeds as in
"certificates up-front": it requests client certificates, and sends a
two-certificate chain. In either case, once the responder has sent its
certificate or certificates, the initiator counts them. If two
certificates have been sent, it proceeds as in "certificates up-front";
otherwise, it proceeds as in "renegotiation" or "in-protocol".

To decide whether to do "renegotiation" or "in-protocol", the
initiator checks whether the responder's initial certificate matches
the criteria listed above.

All new relay implementations of the Tor protocol MUST support
backwards-compatible renegotiation; clients SHOULD do this too. If
this is not possible, new client implementations MUST support both
"renegotiation" and "in-protocol" and use the router's
published link protocols list (see dir-spec.txt on the "protocols" entry)
to decide which to use.

In all of the above handshake variants, certificates sent in the clear
SHOULD NOT include any strings to identify the host as a Tor relay. In
the "renegotiation" and "backwards-compatible renegotiation" steps, the
initiator SHOULD choose a list of ciphersuites and TLS extensions
to mimic one used by a popular web browser.

Even though the connection protocol is identical, we will think of the
initiator as either an onion router (OR) if it is willing to relay
traffic for other Tor users, or an onion proxy (OP) if it only handles
local requests. Onion proxies SHOULD NOT provide long-term-trackable
identifiers in their handshakes.

In all handshake variants, once all certificates are exchanged, all
parties receiving certificates must confirm that the identity key is as
expected. (When initiating a connection, the expected identity key is
the one given in the directory; when creating a connection because of an
EXTEND cell, the expected identity key is the one given in the cell.) If
the key is not as expected, the party must close the connection.

When connecting to an OR, all parties SHOULD reject the connection if that
OR has a malformed or missing certificate. When accepting an incoming
connection, an OR SHOULD NOT reject incoming connections from parties with
malformed or missing certificates. (However, an OR should not believe
that an incoming connection is from another OR unless the certificates
are present and well-formed.)

[Before version 0.1.2.8-rc, ORs rejected incoming connections from ORs and
OPs alike if their certificates were missing or malformed.]

Once a TLS connection is established, the two sides send cells
(specified below) to one another. Cells are sent serially. Standard
cells are CELL_LEN(link_proto) bytes long, but variable-length cells
also exist; see Section 3. Cells may be sent embedded in TLS records
of any size or divided across TLS records, but the framing of TLS
records MUST NOT leak information about the type or contents of the
cells.

TLS connections are not permanent. Either side MAY close a connection
if there are no circuits running over it and an amount of time
(KeepalivePeriod, defaults to 5 minutes) has passed since the last time
any traffic was transmitted over the TLS connection. Clients SHOULD
also hold a TLS connection with no circuits open, if it is likely that a
circuit will be built soon using that connection.

Client-only Tor instances are encouraged to avoid using handshake variants that include certificates, if those certificates provide any persistent tags to the relays they contact. If clients do use certificates, they SHOULD NOT keep using the same certificates when their IP address changes. Clients MAY send certificates using any of the above handshake variants.

2.1. Picking TLS ciphersuites

Clients SHOULD send a ciphersuite list chosen to emulate some popular web browser or other program common on the internet. Clients may send the "Fixed Cipheruite List" below. If they do not, they MUST NOT advertise any ciphersuite that they cannot actually support, unless that cipher is one not supported by OpenSSL 1.0.1.

The fixed ciphersuite list is:

```
TLS1_ECDHE_ECDSA_WITH_AES_256_CBC_SHA
TLS1_ECDHE_RSA_WITH_AES_256_CBC_SHA
TLS1_DHE_RSA_WITH_AES_256_SHA
TLS1_DHE_DSS_WITH_AES_256_SHA
TLS1_ECDH_RSA_WITH_AES_256_CBC_SHA
TLS1_ECDH_ECDSA_WITH_AES_256_CBC_SHA
TLS1_RSA_WITH_AES_256_SHA
TLS1_ECDHE_ECDSA_WITH_RC4_128_SHA
TLS1_ECDHE_ECDSA_WITH_AES_128_CBC_SHA
TLS1_ECDHE_RSA_WITH_RC4_128_SHA
TLS1_ECDHE_RSA_WITH_AES_128_CBC_SHA
TLS1_DHE_RSA_WITH_AES_128_SHA
TLS1_DHE_DSS_WITH_AES_128_SHA
TLS1_ECDH_RSA_WITH_RC4_128_SHA
TLS1_ECDH_RSA_WITH_AES_128_CBC_SHA
TLS1_ECDH_ECDSA_WITH_RC4_128_SHA
TLS1_ECDH_ECDSA_WITH_AES_128_CBC_SHA
SSL3_RSA_RC4_128_MD5
SSL3_RSA_RC4_128_SHA
TLS1_RSA_WITH_AES_128_SHA
TLS1_ECDHE_ECDSA_WITH_DES_192_CBC3_SHA
TLS1_ECDHE_RSA_WITH_DES_192_CBC3_SHA
SSL3_EDH_RSA_DES_192_CBC3_SHA
SSL3_EDH_DSS_DES_192_CBC3_SHA
TLS1_ECDH_RSA_WITH_DES_192_CBC3_SHA
TLS1_ECDH_ECDSA_WITH_DES_192_CBC3_SHA
SSL3_RSA_FIPS_WITH_3DES_EDE_CBC_SHA
SSL3_RSA_DES_192_CBC3_SHA
```

[*] The "extended renegotiation is supported" ciphersuite, 0x00ff, is not counted when checking the list of ciphersuites.

If the client sends the Fixed Ciphersuite List, the responder MUST NOT select any ciphersuite besides TLS_DHE_RSA_WITH_AES_256_CBC_SHA, TLS_DHE_RSA_WITH_AES_128_CBC_SHA, and SSL_DHE_RSA_WITH_3DES_EDE_CBC_SHA: such ciphers might not actually be supported by the client.

If the client sends a v2+ ClientHello with a list of ciphers other then the Fixed Ciphersuite List, the responder can trust that the client supports every cipher advertised in that list, so long as that ciphersuite is also supported by OpenSSL 1.0.1.

Responders MUST NOT select any TLS ciphersuite that lacks ephemeral keys, or whose symmetric keys are less then KEY_LEN bits, or whose digests are less than HASH_LEN bits. Responders SHOULD NOT select any SSLv3 ciphersuite other than the DHE+3DES suites listed above.

2.2. TLS security considerations

Implementations MUST NOT allow TLS session resumption -- it can exacerbate some attacks (e.g. the "Triple Handshake" attack from Feb 2013), and it plays havoc with forward secrecy guarantees.

3. Cell Packet format

The basic unit of communication for onion routers and onion proxies is a fixed-width "cell".

On a version 1 connection, each cell contains the following fields:

```
CircID                                  [CIRCID_LEN bytes]
Command                                 [1 byte]
Payload (padded with 0 bytes)           [PAYLOAD_LEN bytes]
```

On a version 2 or higher connection, all cells are as in version 1 connections, except for variable-length cells, whose format is:

```
CircID                                  [CIRCID_LEN octets]
Command                                 [1 octet]
Length                                  [2 octets; big-endian integer]
Payload                                 [Length bytes]
```

On a version 2 connection, variable-length cells are indicated by a command byte equal to 7 ("VERSIONS"). On a version 3 or higher connection, variable-length cells are indicated by a command byte equal to 7 ("VERSIONS"), or greater than or equal to 128.

CIRCID_LEN is 2 for link protocol versions 1, 2, and 3. CIRCID_LEN is 4 for link protocol version 4 or higher. The VERSIONS cell itself always has CIRCID_LEN == 2 for backward compatibility.

The CircID field determines which circuit, if any, the cell is associated with.

The 'Command' field of a fixed-length cell holds one of the following values:

```
0 -- PADDING     (Padding)                 (See Sec 7.2)
1 -- CREATE      (Create a circuit)        (See Sec 5.1)
```

```
      2 -- CREATED     (Acknowledge create)       (See Sec 5.1)
      3 -- RELAY       (End-to-end data)          (See Sec 5.5 and 6)
      4 -- DESTROY     (Stop using a circuit)     (See Sec 5.4)
      5 -- CREATE_FAST (Create a circuit, no PK)  (See Sec 5.1)
      6 -- CREATED_FAST (Circuit created, no PK)  (See Sec 5.1)
      8 -- NETINFO     (Time and address info)    (See Sec 4.5)
      9 -- RELAY_EARLY (End-to-end data; limited)(See Sec 5.6)
      10 -- CREATE2    (Extended CREATE cell)     (See Sec 5.1)
      11 -- CREATED2   (Extended CREATED cell)     (See Sec 5.1)

   Variable-length command values are:
      7 -- VERSIONS    (Negotiate proto version) (See Sec 4)
      128 -- VPADDING  (Variable-length padding) (See Sec 7.2)
      129 -- CERTS     (Certificates)            (See Sec 4.2)
      130 -- AUTH_CHALLENGE (Challenge value)    (See Sec 4.3)
      131 -- AUTHENTICATE (Client authentication)(See Sec 4.5)
      132 -- AUTHORIZE (Client authorization)    (Not yet used)

   The interpretation of 'Payload' depends on the type of the cell.
      PADDING: Payload is unused.
      CREATE:  Payload contains the handshake challenge.
      CREATED: Payload contains the handshake response.
      RELAY:   Payload contains the relay header and relay body.
      DESTROY: Payload contains a reason for closing the circuit.
               (see 5.4)
   Upon receiving any other value for the command field, an OR must
   drop the cell.  Since more cell types may be added in the future, ORs
   should generally not warn when encountering unrecognized commands.

   The payload is padded with 0 bytes.

   PADDING cells are currently used to implement connection keepalive.
   If there is no other traffic, ORs and OPs send one another a PADDING
   cell every few minutes.

   CREATE, CREATED, and DESTROY cells are used to manage circuits;
   see section 5 below.

   RELAY cells are used to send commands and data along a circuit; see
   section 6 below.

   VERSIONS and NETINFO cells are used to set up connections in link
   protocols v2 and higher; in link protocol v3 and higher, CERTS,
   AUTH_CHALLENGE, and AUTHENTICATE may also be used.  See section 4
   below.

4. Negotiating and initializing connections

   After Tor instances negotiate handshake with either the "renegotiation" or
   "in-protocol" handshakes, they must exchange a set of cells to set up
   the Tor connection and make it "open" and usable for circuits.

   When the renegotiation handshake is used, both parties immediately
   send a VERSIONS cell (4.1 below), and after negotiating a link
   protocol version (which will be 2), each send a NETINFO cell (4.5
   below) to confirm their addresses and timestamps.  No other intervening
   cell types are allowed.

   When the in-protocol handshake is used, the initiator sends a
   VERSIONS cell to indicate that it will not be renegotiating.  The
   responder sends a VERSIONS cell, a CERTS cell (4.2 below) to give the
   initiator the certificates it needs to learn the responder's
   identity, an AUTH_CHALLENGE cell (4.3) that the initiator must include
   as part of its answer if it chooses to authenticate, and a NETINFO
   cell (4.5).  As soon as it gets the CERTS cell, the initiator knows
   whether the responder is correctly authenticated.  At this point the
   initiator may send a NETINFO cell if it does not wish to
   authenticate, or a CERTS cell, an AUTHENTICATE cell (4.4), and a NETINFO
   cell if it does.  When this handshake is in use, the first cell must
   be VERSIONS, VPADDING or AUTHORIZE, and no other cell type is allowed to
   intervene besides those specified, except for PADDING and VPADDING cells.

   The AUTHORIZE cell type is reserved for future use by scanning-resistance
   designs.

   [Tor versions before 0.2.3.11-alpha did not recognize the AUTHORIZE cell,
   and did not permit any command other than VERSIONS as the first cell of
   the in-protocol handshake.]

4.1. Negotiating versions with VERSIONS cells

   There are multiple instances of the Tor link connection protocol.  Any
   connection negotiated using the "certificates up front" handshake (see
   section 2 above) is "version 1".  In any connection where both parties
   have behaved as in the "renegotiation" handshake, the link protocol
   version must be 2.  In any connection where both parties have behaved
   as in the "in-protocol" handshake, the link protocol must be 3 or higher.

   To determine the version, in any connection where the "renegotiation"
   or "in-protocol" handshake was used (that is, where the responder
   sent only one certificate at first and where the initiator did not
   send any certificates in the first negotiation), both parties MUST
   send a VERSIONS cell.  In "renegotiation", they send a VERSIONS cell
   right after the renegotiation is finished, before any other cells are
   sent.  In "in-protocol", the initiator sends a VERSIONS cell
   immediately after the initial TLS handshake, and the responder
   replies immediately with a VERSIONS cell.  Parties MUST NOT send any
   other cells on a connection until they have received a VERSIONS cell.

   The payload in a VERSIONS cell is a series of big-endian two-byte
   integers.  Both parties MUST select as the link protocol version the
   highest number contained both in the VERSIONS cell they sent and in the
   versions cell they received.  If they have no such version in common,
   they cannot communicate and MUST close the connection.  Either party MUST
   close the connection if the versions cell is not well-formed (for example,
```

if it contains an odd number of bytes).

Since the version 1 link protocol does not use the "renegotiation" handshake, implementations MUST NOT list version 1 in their VERSIONS cell. When the "renegotiation" handshake is used, implementations MUST list only the version 2. When the "in-protocol" handshake is used, implementations MUST NOT list any version before 3, and SHOULD list at least version 3.

Link protocols differences are:
1 -- The "certs up front" handshake.
2 -- Uses the renegotiation-based handshake. Introduces variable-length cells.
3 -- Uses the in-protocol handshake.
4 -- Increases circuit ID width to 4 bytes.

4.2. CERTS cells

The CERTS cell describes the keys that a Tor instance is claiming to have. It is a variable-length cell. Its payload format is:

```
N: Number of certs in cell            [1 octet]
N times:
   CertType                           [1 octet]
   CLEN                               [2 octets]
   Certificate                        [CLEN octets]
```

Any extra octets at the end of a CERTS cell MUST be ignored.

CertType values are:
1: Link key certificate certified by RSA1024 identity
2: RSA1024 Identity certificate
3: RSA1024 AUTHENTICATE cell link certificate

The certificate format for the above certificate types is DER encoded X509.

A CERTS cell may have no more than one certificate of each CertType.

To authenticate the responder, the initiator MUST check the following:
* The CERTS cell contains exactly one CertType 1 "Link" certificate.
* The CERTS cell contains exactly one CertType 2 "ID" certificate.
* Both certificates have validAfter and validUntil dates that are not expired.
* The certified key in the Link certificate matches the link key that was used to negotiate the TLS connection.
* The certified key in the ID certificate is a 1024-bit RSA key.
* The certified key in the ID certificate was used to sign both certificates.
* The link certificate is correctly signed with the key in the ID certificate
* The ID certificate is correctly self-signed.

Checking these conditions is sufficient to authenticate that the initiator is talking to the Tor node with the expected identity, as certified in the ID certificate.

To authenticate the initiator, the responder MUST check the following:
* The CERTS cell contains exactly one CertType 3 "AUTH" certificate.
* The CERTS cell contains exactly one CertType 2 "ID" certificate.
* Both certificates have validAfter and validUntil dates that are not expired.
* The certified key in the AUTH certificate is a 1024-bit RSA key.
* The certified key in the ID certificate is a 1024-bit RSA key.
* The certified key in the ID certificate was used to sign both certificates.
* The auth certificate is correctly signed with the key in the ID certificate.
* The ID certificate is correctly self-signed.

Checking these conditions is NOT sufficient to authenticate that the initiator has the ID it claims; to do so, the cells in 4.3 and 4.4 below must be exchanged.

4.3. AUTH_CHALLENGE cells

An AUTH_CHALLENGE cell is a variable-length cell with the following fields:

```
Challenge [32 octets]
N_Methods [2 octets]
Methods   [2 * N_Methods octets]
```

It is sent from the responder to the initiator. Initiators MUST ignore unexpected bytes at the end of the cell. Responders MUST generate every challenge independently using a strong RNG or PRNG.

The Challenge field is a randomly generated string that the initiator must sign (a hash of) as part of authenticating. The methods are the authentication methods that the responder will accept. Only one authentication method is defined right now: see 4.4 below.

4.4. AUTHENTICATE cells

If an initiator wants to authenticate, it responds to the AUTH_CHALLENGE cell with a CERTS cell and an AUTHENTICATE cell. The CERTS cell is as a server would send, except that instead of sending a CertType 1 cert for an arbitrary link certificate, the client sends a CertType 3 cert for an RSA AUTHENTICATE key. (This difference is because we allow any link key type on a TLS link, but the protocol described here will only work for 1024-bit RSA keys. A later protocol version should extend the protocol here to work with non-1024-bit, non-RSA keys.)

An AUTHENTICATE cell contains the following:

```
      AuthType                              [2 octets]
      AuthLen                               [2 octets]
      Authentication                        [AuthLen octets]
```

Responders MUST ignore extra bytes at the end of an AUTHENTICATE cell. If AuthType is 1 (meaning "RSA-SHA256-TLSSecret"), then the Authentication contains the following:

```
    TYPE: The characters "AUTH0001" [8 octets]
    CID: A SHA256 hash of the initiator's RSA1024 identity key [32 octets]
    SID: A SHA256 hash of the responder's RSA1024 identity key [32 octets]
    SLOG: A SHA256 hash of all bytes sent from the responder to the
      initiator as part of the negotiation up to and including the
      AUTH_CHALLENGE cell; that is, the VERSIONS cell, the CERTS cell,
      the AUTH_CHALLENGE cell, and any padding cells.  [32 octets]
    CLOG: A SHA256 hash of all bytes sent from the initiator to the
      responder as part of the negotiation so far; that is, the
      VERSIONS cell and the CERTS cell and any padding cells. [32
      octets]
    SCERT: A SHA256 hash of the responder's TLS link certificate. [32
      octets]
    TLSSECRETS: A SHA256 HMAC, using the TLS master secret as the
      secret key, of the following:
        - client_random, as sent in the TLS Client Hello
        - server_random, as sent in the TLS Server Hello
        - the NUL terminated ASCII string:
          "Tor V3 handshake TLS cross-certification"
       [32 octets]
    RAND: A 24 byte value, randomly chosen by the initiator.  (In an
      imitation of SSL3's gmt_unix_time field, older versions of Tor
      sent an 8-byte timestamp as the first 8 bytes of this field;
      new implementations should not do that.) [24 octets]
    SIG: A signature of a SHA256 hash of all the previous fields
      using the initiator's "Authenticate" key as presented.  (As
      always in Tor, we use OAEP-MGF1 padding; see tor-spec.txt
      section 0.3.)
       [variable length]
```

To check the AUTHENTICATE cell, a responder checks that all fields from TYPE through TLSSECRETS contain their unique correct values as described above, and then verifies the signature. The server MUST ignore any extra bytes in the signed data after the SHA256 hash.

Initiators MUST NOT send an AUTHENTICATE cell before they have verified the certificates presented in the responder's CERTS cell, and authenticated the responder.

4.5. NETINFO cells

If version 2 or higher is negotiated, each party sends the other a NETINFO cell. The cell's payload is:

```
      Timestamp              [4 bytes]
      Other OR's address     [variable]
      Number of addresses    [1 byte]
      This OR's addresses    [variable]
```

The address format is a type/length/value sequence as given in section 6.4 below. The timestamp is a big-endian unsigned integer number of seconds since the Unix epoch.

Implementations MAY use the timestamp value to help decide if their clocks are skewed. Initiators MAY use "other OR's address" to help learn which address their connections are originating from, if they do not know it. [As of 0.2.3.1-alpha, nodes use neither of these values.]

Initiators SHOULD use "this OR's address" to make sure that they have connected to another OR at its canonical address. (See 5.3.1 below.)

5. Circuit management

5.1. CREATE and CREATED cells

Users set up circuits incrementally, one hop at a time. To create a new circuit, OPs send a CREATE cell to the first node, with the first half of an authenticated handshake; that node responds with a CREATED cell with the second half of the handshake. To extend a circuit past the first hop, the OP sends an EXTEND relay cell (see section 5.1.2) which instructs the last node in the circuit to send a CREATE cell to extend the circuit.

There are two kinds of CREATE and CREATED cells: The older "CREATE/CREATED" format, and the newer "CREATE2/CREATED2" format. The newer format is extensible by design; the older one is not.

A CREATE2 cell contains:

```
    HTYPE     (Client Handshake Type)     [2 bytes]
    HLEN      (Client Handshake Data Len) [2 bytes]
    HDATA     (Client Handshake Data)     [HLEN bytes]
```

A CREATED2 cell contains:

```
    HLEN      (Server Handshake Data Len) [2 bytes]
    HDATA     (Server Handshake Data)     [HLEN bytes]
```

Recognized handshake types are:

```
    0x0000  TAP  -- the original Tor handshake; see 5.1.3
    0x0001  reserved
    0x0002  ntor -- the ntor+curve25519+sha256 handshake; see 5.1.4
```

The format of a CREATE cell is one of the following:

```
    HDATA     (Client Handshake Data)     [TAP_C_HANDSHAKE_LEN bytes]
```

--

or
    HTAG   (Client Handshake Type Tag) [16 bytes]
    HDATA   (Client Handshake Data)   [TAP_C_HANDSHAKE_LEN-16 bytes]

The first format is equivalent to a CREATE2 cell with HTYPE of 'tap'
and length of TAP_C_HANDSHAKE_LEN. The second format is a way to
encapsulate new handshake types into the old CREATE cell format for
migration. See 5.1.2.1 below. Recognized HTAG values are:

    ntor -- 'ntorNTORntorNTOR'

The format of a CREATED cell is:
    HDATA   (Server Handshake Data)   [TAP_S_HANDSHAKE_LEN bytes]
(It's equivalent to a CREATED2 cell with length of TAP_S_HANDSHAKE_LEN.)

As usual with DH, x and y MUST be generated randomly.

In general, clients SHOULD use CREATE whenever they are using the TAP
handshake, and CREATE2 otherwise. Clients SHOULD NOT send the
second format of CREATE cells (the one with the handshake type tag)
to a server directly.

Servers always reply to a successful CREATE with a CREATED, and to a
successful CREATE2 with a CREATED2. On failure, a server sends a
DESTROY cell to tear down the circuit.

[CREATE2 is handled by Tor 0.2.4.7-alpha and later.]

5.1.1. Choosing circuit IDs in create cells

The CircID for a CREATE cell is an arbitrarily chosen nonzero integer,
selected by the node (OP or OR) that sends the CREATE cell. In link
protocol 3 or lower, CircIDs are 2 bytes long; in protocol 4 or
higher, CircIDs are 4 bytes long.

To prevent CircID collisions, when one node sends a CREATE cell to
another, it chooses from only one half of the possible values based
on the ORs' public identity keys. In link protocol version 3 or
lower, if the sending node has a lower key, it chooses a CircID with
an MSB of 0; otherwise, it chooses a CircID with an MSB of 1. (Public
keys are compared numerically by modulus.)

In link protocol version 4 or higher, whichever node initiated the
connection sets its MSB to 1, and whichever node didn't initiate the
connection sets its MSB to 0.

(An OP with no public key MAY choose any CircID it wishes, since an OP
never needs to process a CREATE cell.)

The CircID value 0 is specifically reserved for cells that do not
belong to any circuit: CircID 0 must not be used for circuits. No
other CircID value, including 0x8000 or 0x80000000, is reserved.

5.1.2. EXTEND and EXTENDED cells

To extend an existing circuit, the client sends a EXTEND or EXTENDED2
relay cell to the last node in the circuit.

An EXTEND2 cell's relay payload contains:
    NSPEC   (Number of link specifiers)   [1 byte]
      NSPEC times:
        LSTYPE (Link specifier type)   [1 byte]
        LSLEN (Link specifier length)   [1 byte]
        LSPEC (Link specifier)   [LSLEN bytes]
    HTYPE   (Client Handshake Type)   [2 bytes]
    HLEN   (Client Handshake Data Len)   [2 bytes]
    HDATA   (Client Handshake Data)   [HLEN bytes]

Link specifiers describe the next node in the circuit and how to
connect to it. Recognized specifiers are:
   [00] TLS-over-TCP, IPv4 address
        A four-byte IPv4 address plus two-byte ORPort
   [01] TLS-over-TCP, IPv6 address
        A sixteen-byte IPv6 address plus two-byte ORPort
   [02] Legacy identity
        A 20-byte SHA1 identity fingerprint. At most one may be listed.

Nodes MUST ignore unrecognized specifiers, and MUST accept multiple
instances of specifiers other than 'legacy identity'.

The relay payload for an EXTEND relay cell consists of:
      Address   [4 bytes]
      Port   [2 bytes]
      Onion skin   [TAP_C_HANDSHAKE_LEN bytes]
      Identity fingerprint   [HASH_LEN bytes]

The "legacy identity" and "identity fingerprint fields are the SHA1
hash of the PKCS#1 ASN1 encoding of the next onion router's identity
(signing) key. (See 0.3 above.) Including this hash allows the
extending OR verify that it is indeed connected to the correct target
OR, and prevents certain man-in-the-middle attacks.

The payload of an EXTENDED cell is the same as the payload of a
CREATED cell.

The payload of an EXTENDED2 cell is the same as the payload of a
CREATED2 cell.

[Support for EXTEND2 was added in Tor 0.2.4.8-alpha.]

Clients SHOULD use the EXTEND format whenever sending a TAP
handshake, and MUST use it whenever the EXTEND cell will be handled
by a node running a version of Tor too old to support EXTEND2. In
other cases, clients SHOULD use EXTEND2.

When encoding a non-TAP handshake in an EXTEND cell, clients SHOULD

When encoding a new TAP handshake in an EXTEND cell, clients should use the format with 'client handshake type tag'.

5.1.3. The "TAP" handshake

This handshake uses Diffie-Hellman in Z_p and RSA to compute a set of shared keys which the client knows are shared only with a particular server, and the server knows are shared with whomever sent the original handshake (or with nobody at all). It's not very fast and not very good. (See Goldberg's "On the Security of the Tor Authentication Protocol".)

Define TAP_C_HANDSHAKE_LEN as DH_LEN+KEY_LEN+PK_PAD_LEN.
Define TAP_S_HANDSHAKE_LEN as DH_LEN+HASH_LEN.

The payload for a CREATE cell is an 'onion skin', which consists of the first step of the DH handshake data (also known as g^x). This value is hybrid-encrypted (see 0.3) to the server's onion key, giving a client handshake of:

```
    PK-encrypted:
      Padding                       [PK_PAD_LEN bytes]
      Symmetric key                 [KEY_LEN bytes]
      First part of g^x             [PK_ENC_LEN-PK_PAD_LEN-KEY_LEN bytes]
    Symmetrically encrypted:
      Second part of g^x            [DH_LEN-(PK_ENC_LEN-PK_PAD_LEN-KEY_LEN)
                                        bytes]
```

The payload for a CREATED cell, or the relay payload for an EXTENDED cell, contains:

```
      DH data (g^y)                 [DH_LEN bytes]
      Derivative key data (KH)      [HASH_LEN bytes]   <see 5.2 below>
```

Once the handshake between the OP and an OR is completed, both can now calculate g^xy with ordinary DH. Before computing g^xy, both parties MUST verify that the received g^x or g^y value is not degenerate; that is, it must be strictly greater than 1 and strictly less than p-1 where p is the DH modulus. Implementations MUST NOT complete a handshake with degenerate keys. Implementations MUST NOT discard other "weak" g^x values.

(Discarding degenerate keys is critical for security; if bad keys are not discarded, an attacker can substitute the OR's CREATED cell's g^y with 0 or 1, thus creating a known g^xy and impersonating the OR. Discarding other keys may allow attacks to learn bits of the private key.)

Once both parties have g^xy, they derive their shared circuit keys and 'derivative key data' value via the KDF-TOR function in 5.2.1.

5.1.4. The "ntor" handshake

This handshake uses a set of DH handshakes to compute a set of shared keys which the client knows are shared only with a particular server, and the server knows are shared with whomever sent the original handshake (or with nobody at all). Here we use the "curve25519" group and representation as specified in "Curve25519: new Diffie-Hellman speed records" by D. J. Bernstein.

[The ntor handshake was added in Tor 0.2.4.8-alpha.]

In this section, define:

```
   H(x,t) as HMAC_SHA256 with message x and key t.
   H_LENGTH  = 32.
   ID_LENGTH = 20.
   G_LENGTH  = 32
   PROTOID   = "ntor-curve25519-sha256-1"
   t_mac     = PROTOID | ":mac"
   t_key     = PROTOID | ":key_extract"
   t_verify  = PROTOID | ":verify"
   MULT(a,b) = the multiplication of the curve25519 point 'a' by the
               scalar 'b'.
   G         = The preferred base point for curve25519 ([9])
   KEYGEN()  = The curve25519 key generation algorithm, returning
               a private/public keypair.
   m_expand  = PROTOID | ":key_expand"
   KEYID(A)  = A
```

To perform the handshake, the client needs to know an identity key digest for the server, and an ntor onion key (a curve25519 public key) for that server. Call the ntor onion key "B". The client generates a temporary keypair:

```
    x,X = KEYGEN()
```

and generates a client-side handshake with contents:

```
    NODEID      Server identity digest  [ID_LENGTH bytes]
    KEYID       KEYID(B)                [H_LENGTH bytes]
    CLIENT_PK   X                       [G_LENGTH bytes]
```

The server generates a keypair of y,Y = KEYGEN(), and uses its ntor private key 'b' to compute:

```
  secret_input = EXP(X,y) | EXP(X,b) | ID | B | X | Y | PROTOID
  KEY_SEED = H(secret_input, t_key)
  verify = H(secret_input, t_verify)
  auth_input = verify | ID | B | Y | X | PROTOID | "Server"
```

The server's handshake reply is:

```
    SERVER_PK   Y                       [G_LENGTH bytes]
    AUTH        H(auth_input, t_mac)    [H_LENGTH bytes]
```

The client then checks Y is in G^* [see NOTE below], and computes

```
  secret_input = EXP(Y,x) | EXP(B,x) | ID | B | X | Y | PROTOID
  KEY_SEED = H(secret_input, t_key)
  verify = H(secret_input, t_verify)
  auth_input = verify | ID | B | Y | X | PROTOID | "Server"
```

The client verifies that AUTH == H(auth_input, t_mac).

Both parties check that none of the EXP() operations produced the point at infinity. [NOTE: This is an adequate replacement for checking Y for group membership, if the group is curve25519.]

Both parties now have a shared value for KEY_SEED. They expand this into the keys needed for the Tor relay protocol, using the KDF described in 5.2.2 and the tag m_expand.

5.1.5. CREATE_FAST/CREATED_FAST cells

When initializing the first hop of a circuit, the OP has already established the OR's identity and negotiated a secret key using TLS. Because of this, it is not always necessary for the OP to perform the public key operations to create a circuit. In this case, the OP MAY send a CREATE_FAST cell instead of a CREATE cell for the first hop only. The OR responds with a CREATED_FAST cell, and the circuit is created.

A CREATE_FAST cell contains:

```
Key material (X)    [HASH_LEN bytes]
```

A CREATED_FAST cell contains:

```
Key material (Y)    [HASH_LEN bytes]
Derivative key data [HASH_LEN bytes] (See 5.2.1 below)
```

The values of X and Y must be generated randomly.

Once both parties have X and Y, they derive their shared circuit keys and 'derivative key data' value via the KDF-TOR function in 5.2.1.

If an OR sees a circuit created with CREATE_FAST, the OR is sure to be the first hop of a circuit. ORs SHOULD reject attempts to create streams with RELAY_BEGIN exiting the circuit at the first hop: letting Tor be used as a single hop proxy makes exit nodes a more attractive target for compromise.

The CREATE_FAST handshake is currently deprecated whenever it is not necessary; the migration is controlled by the "usecreatefast" networkstatus parameter as described in dir-spec.txt.

5.2. Setting circuit keys

5.2.1. KDF-TOR

This key derivation function is used by the TAP and CREATE_FAST handshakes, and in the current hidden service protocol. It shouldn't be used for new functionality.

If the TAP handshake is used to extend a circuit, both parties base their key material on K0=g^xy, represented as a big-endian unsigned integer.

If CREATE_FAST is used, both parties base their key material on K0=X|Y.

From the base key material K0, they compute KEY_LEN*2+HASH_LEN*3 bytes of derivative key data as

```
K = H(K0 | [00]) | H(K0 | [01]) | H(K0 | [02]) | ...
```

The first HASH_LEN bytes of K form KH; the next HASH_LEN form the forward digest Df; the next HASH_LEN 41-60 form the backward digest Db; the next KEY_LEN 61-76 form Kf, and the final KEY_LEN form Kb. Excess bytes from K are discarded.

KH is used in the handshake response to demonstrate knowledge of the computed shared key. Df is used to seed the integrity-checking hash for the stream of data going from the OP to the OR, and Db seeds the integrity-checking hash for the data stream from the OR to the OP. Kf is used to encrypt the stream of data going from the OP to the OR, and Kb is used to encrypt the stream of data going from the OR to the OP.

5.2.2. KDF-RFC5869

For newer KDF needs, Tor uses the key derivation function HKDF from RFC5869, instantiated with SHA256. (This is due to a construction from Krawczyk.) The generated key material is:

```
K = K_1 | K_2 | K_3 | ...

Where H(x,t) is HMAC_SHA256 with value x and key t
  and K_1     = H(m_expand | INT8(1) , KEY_SEED )
  and K_(i+1) = H(K_i | m_expand | INT8(i+1) , KEY_SEED )
  and m_expand is an arbitrarily chosen value,
  and INT8(i) is a octet with the value "i".
```

In RFC5869's vocabulary, this is HKDF-SHA256 with info == m_expand, salt == t_key, and IKM == secret_input.

When used in the ntor handshake, the first HASH_LEN bytes form the forward digest Df; the next HASH_LEN form the backward digest Db; the next KEY_LEN form Kf, the next KEY_LEN form Kb, and the final DIGEST_LEN bytes are taken as a nonce to use in the place of KH in the hidden service protocol. Excess bytes from K are discarded.

5.3. Creating circuits

When creating a circuit through the network, the circuit creator (OP) performs the following steps:

1. Choose an onion router as an exit node (R_N), such that the onion router's exit policy includes at least one pending stream that

needs a circuit (if there are any).

2. Choose a chain of (N-1) onion routers
(R_1...R_N-1) to constitute the path, such that no router
appears in the path twice.

3. If not already connected to the first router in the chain,
open a new connection to that router.

4. Choose a circID not already in use on the connection with the
first router in the chain; send a CREATE cell along the
connection, to be received by the first onion router.

5. Wait until a CREATED cell is received; finish the handshake
and extract the forward key Kf_1 and the backward key Kb_1.

6. For each subsequent onion router R (R_2 through R_N), extend
the circuit to R.

To extend the circuit by a single onion router R_M, the OP performs
these steps:

1. Create an onion skin, encrypted to R_M's public onion key.

2. Send the onion skin in a relay EXTEND cell along
the circuit (see section 5).

3. When a relay EXTENDED cell is received, verify KH, and
calculate the shared keys. The circuit is now extended.

When an onion router receives an EXTEND relay cell, it sends a CREATE
cell to the next onion router, with the enclosed onion skin as its
payload. As special cases, if the extend cell includes a digest of
all zeroes, or asks to extend back to the relay that sent the extend
cell, the circuit will fail and be torn down. The initiating onion
router chooses some circID not yet used on the connection between the
two onion routers. (But see section 5.1.1 above, concerning choosing
circIDs based on lexicographic order of nicknames.)

When an onion router receives a CREATE cell, if it already has a
circuit on the given connection with the given circID, it drops the
cell. Otherwise, after receiving the CREATE cell, it completes the
DH handshake, and replies with a CREATED cell. Upon receiving a
CREATED cell, an onion router packs it payload into an EXTENDED relay
cell (see section 5), and sends that cell up the circuit. Upon
receiving the EXTENDED relay cell, the OP can retrieve g^y.

(As an optimization, OR implementations may delay processing onions
until a break in traffic allows time to do so without harming
network latency too greatly.)

5.3.1. Canonical connections

It is possible for an attacker to launch a man-in-the-middle attack
against a connection by telling OR Alice to extend to OR Bob at some
address X controlled by the attacker. The attacker cannot read the
encrypted traffic, but the attacker is now in a position to count all
bytes sent between Alice and Bob (assuming Alice was not already
connected to Bob.)

To prevent this, when an OR gets an extend request, it SHOULD use an
existing OR connection if the ID matches, and ANY of the following
conditions hold:
- The IP matches the requested IP.
- The OR knows that the IP of the connection it's using is canonical
because it was listed in the NETINFO cell.
- The OR knows that the IP of the connection it's using is canonical
because it was listed in the server descriptor.

[This is not implemented in Tor 0.2.0.23-rc.]

5.4. Tearing down circuits

Circuits are torn down when an unrecoverable error occurs along
the circuit, or when all streams on a circuit are closed and the
circuit's intended lifetime is over. Circuits may be torn down
either completely or hop-by-hop.

To tear down a circuit completely, an OR or OP sends a DESTROY
cell to the adjacent nodes on that circuit, using the appropriate
direction's circID.

Upon receiving an outgoing DESTROY cell, an OR frees resources
associated with the corresponding circuit. If it's not the end of
the circuit, it sends a DESTROY cell for that circuit to the next OR
in the circuit. If the node is the end of the circuit, then it tears
down any associated edge connections (see section 6.1).

After a DESTROY cell has been processed, an OR ignores all data or
destroy cells for the corresponding circuit.

To tear down part of a circuit, the OP may send a RELAY_TRUNCATE cell
signaling a given OR (Stream ID zero). That OR sends a DESTROY
cell to the next node in the circuit, and replies to the OP with a
RELAY_TRUNCATED cell.

[Note: If an OR receives a TRUNCATE cell and it has any RELAY cells
still queued on the circuit for the next node it will drop them
without sending them. This is not considered conformant behavior,
but it probably won't get fixed until a later version of Tor. Thus,
clients SHOULD NOT send a TRUNCATE cell to a node running any current
version of Tor if a) they have sent relay cells through that node,
and b) they aren't sure whether those cells have been sent on yet.]

When an unrecoverable error occurs along one connection in a

circuit, the nodes on either side of the connection should, if they
are able, act as follows: the node closer to the OP should send a
RELAY_TRUNCATED cell towards the OP; the node farther from the OP
should send a DESTROY cell down the circuit.

The payload of a RELAY_TRUNCATED or DESTROY cell contains a single octet,
describing why the circuit is being closed or truncated. When sending a
TRUNCATED or DESTROY cell because of another TRUNCATED or DESTROY cell,
the error code should be propagated. The origin of a circuit always sets
this error code to 0, to avoid leaking its version.

The error codes are:

```
  0 -- NONE             (No reason given.)
  1 -- PROTOCOL         (Tor protocol violation.)
  2 -- INTERNAL         (Internal error.)
  3 -- REQUESTED        (A client sent a TRUNCATE command.)
  4 -- HIBERNATING      (Not currently operating; trying to save bandwidth.)
  5 -- RESOURCELIMIT    (Out of memory, sockets, or circuit IDs.)
  6 -- CONNECTFAILED    (Unable to reach relay.)
  7 -- OR_IDENTITY      (Connected to relay, but its OR identity was not
                         as expected.)
  8 -- OR_CONN_CLOSED   (The OR connection that was carrying this circuit
                         died.)
  9 -- FINISHED         (The circuit has expired for being dirty or old.)
 10 -- TIMEOUT          (Circuit construction took too long)
 11 -- DESTROYED        (The circuit was destroyed w/o client TRUNCATE)
 12 -- NOSUCHSERVICE    (Request for unknown hidden service)
```

5.5. Routing relay cells

When an OR receives a RELAY or RELAY_EARLY cell, it checks the cell's
circID and determines whether it has a corresponding circuit along that
connection. If not, the OR drops the cell.

Otherwise, if the OR is not at the OP edge of the circuit (that is,
either an 'exit node' or a non-edge node), it de/encrypts the payload
with the stream cipher, as follows:

```
     'Forward' relay cell (same direction as CREATE):
        Use Kf as key; decrypt.
     'Back' relay cell (opposite direction from CREATE):
        Use Kb as key; encrypt.
```

Note that in counter mode, decrypt and encrypt are the same operation.

The OR then decides whether it recognizes the relay cell, by
inspecting the payload as described in section 6.1 below. If the OR
recognizes the cell, it processes the contents of the relay cell.
Otherwise, it passes the decrypted relay cell along the circuit if
the circuit continues. If the OR at the end of the circuit
encounters an unrecognized relay cell, an error has occurred: the OR
sends a DESTROY cell to tear down the circuit.

When a relay cell arrives at an OP, the OP decrypts the payload
with the stream cipher as follows:

```
      OP receives data cell:
         For I=N...1,
             Decrypt with Kb_I.  If the payload is recognized (see
             section 6..1), then stop and process the payload.
```

For more information, see section 6 below.

5.6. Handling relay_early cells

A RELAY_EARLY cell is designed to limit the length any circuit can reach.
When an OR receives a RELAY_EARLY cell, and the next node in the circuit
is speaking v2 of the link protocol or later, the OR relays the cell as a
RELAY_EARLY cell. Otherwise, older Tors will relay it as a RELAY cell.

If a node ever receives more than 8 RELAY_EARLY cells on a given
outbound circuit, it SHOULD close the circuit. If it receives any
inbound RELAY_EARLY cells, it MUST close the circuit immediately.

When speaking v2 of the link protocol or later, clients MUST only send
EXTEND cells inside RELAY_EARLY cells. Clients SHOULD send the first ~8
RELAY cells that are not targeted at the first hop of any circuit as
RELAY_EARLY cells too, in order to partially conceal the circuit length.

[Starting with Tor 0.2.3.11-alpha, relays should
reject any EXTEND cell not received in a RELAY_EARLY cell.]

6. Application connections and stream management

6.1. Relay cells

Within a circuit, the OP and the exit node use the contents of
RELAY packets to tunnel end-to-end commands and TCP connections
("Streams") across circuits. End-to-end commands can be initiated
by either edge; streams are initiated by the OP.

The payload of each unencrypted RELAY cell consists of:

```
      Relay command           [1 byte]
      'Recognized'            [2 bytes]
      StreamID                [2 bytes]
      Digest                  [4 bytes]
      Length                  [2 bytes]
      Data                    [PAYLOAD_LEN-11 bytes]
```

The relay commands are:

```
      1 -- RELAY_BEGIN     [forward]
      2 -- RELAY_DATA      [forward or backward]
      3 -- RELAY_END       [forward or backward]
      4 -- RELAY_CONNECTED [backward]
      5 -- RELAY_SENDME    [forward or backward] [sometimes control]
      6 -- RELAY_EXTEND    [forward]             [control]
      7 -- RELAY_EXTENDED  [backward]            [control]
      8 -- RELAY_TRUNCATE  [forward]             [control]
```

```
       9 -- RELAY_TRUNCATED [backward]            [control]
      10 -- RELAY_DROP      [forward or backward] [control]
      11 -- RELAY_RESOLVE   [forward]
      12 -- RELAY_RESOLVED  [backward]
      13 -- RELAY_BEGIN_DIR [forward]
      14 -- RELAY_EXTEND2   [forward]             [control]
      15 -- RELAY_EXTENDED2 [backward]            [control]

     32..40 -- Used for hidden services; see rend-spec.txt.
```

Commands labelled as "forward" must only be sent by the originator of the circuit. Commands labelled as "backward" must only be sent by other nodes in the circuit back to the originator. Commands marked as either can be sent either by the originator or other nodes.

The 'recognized' field in any unencrypted relay payload is always set to zero; the 'digest' field is computed as the first four bytes of the running digest of all the bytes that have been destined for this hop of the circuit or originated from this hop of the circuit, seeded from Df or Db respectively (obtained in section 5.2 above), and including this RELAY cell's entire payload (taken with the digest field set to zero).

When the 'recognized' field of a RELAY cell is zero, and the digest is correct, the cell is considered "recognized" for the purposes of decryption (see section 5.5 above).

(The digest does not include any bytes from relay cells that do not start or end at this hop of the circuit. That is, it does not include forwarded data. Therefore if 'recognized' is zero but the digest does not match, the running digest at that node should not be updated, and the cell should be forwarded on.)

All RELAY cells pertaining to the same tunneled stream have the same stream ID. StreamIDs are chosen arbitrarily by the OP. No stream may have a StreamID of zero. Rather, RELAY cells that affect the entire circuit rather than a particular stream use a StreamID of zero -- they are marked in the table above as "[control]" style cells. (Sendme cells are marked as "sometimes control" because they can include a StreamID or not depending on their purpose -- see Section 7.)

The 'Length' field of a relay cell contains the number of bytes in the relay payload which contain real payload data. The remainder of the payload is padded with NUL bytes.

If the RELAY cell is recognized but the relay command is not understood, the cell must be dropped and ignored. Its contents still count with respect to the digests and flow control windows, though.

6.2. Opening streams and transferring data

To open a new anonymized TCP connection, the OP chooses an open circuit to an exit that may be able to connect to the destination address, selects an arbitrary StreamID not yet used on that circuit, and constructs a RELAY_BEGIN cell with a payload encoding the address and port of the destination host. The payload format is:

```
      ADDRPORT [nul-terminated string]
      FLAGS    [4 bytes]
```

ADDRPORT is made of ADDRESS | ':' | PORT | [00]

where ADDRESS can be a DNS hostname, or an IPv4 address in dotted-quad format, or an IPv6 address surrounded by square brackets; and where PORT is a decimal integer between 1 and 65535, inclusive.

The FLAGS value has one or more of the following bits set, where "bit 1" is the LSB of the 32-bit value, and "bit 32" is the MSB. (Remember that all values in Tor are big-endian (see 0.1.1 above), so the MSB of a 4-byte value is the MSB of the first byte, and the LSB of a 4-byte value is the LSB of its last byte.)

```
  bit   meaning
   1 -- IPv6 okay.  We support learning about IPv6 addresses and
        connecting to IPv6 addresses.
   2 -- IPv4 not okay.  We don't want to learn about IPv4 addresses
        or connect to them.
   3 -- IPv6 preferred.  If there are both IPv4 and IPv6 addresses,
        we want to connect to the IPv6 one.  (By default, we connect
        to the IPv4 address.)
  4..32 -- Reserved. Current clients MUST NOT set these. Servers
        MUST ignore them.
```

Upon receiving this cell, the exit node resolves the address as necessary, and opens a new TCP connection to the target port. If the address cannot be resolved, or a connection can't be established, the exit node replies with a RELAY_END cell. (See 6.4 below.) Otherwise, the exit node replies with a RELAY_CONNECTED cell, whose payload is in one of the following formats:

```
    The IPv4 address to which the connection was made [4 octets]
    A number of seconds (TTL) for which the address may be cached [4 octets]
 or
    Four zero-valued octets [4 octets]
    An address type (6)     [1 octet]
    The IPv6 address to which the connection was made [16 octets]
    A number of seconds (TTL) for which the address may be cached [4 octets]
```

[Tor exit nodes before 0.1.2.0 set the TTL field to a fixed value. Later versions set the TTL to the last value seen from a DNS server, and expire their own cached entries after a fixed interval. This prevents certain attacks.]

Once a connection has been established, the OP and exit node package stream data in RELAY_DATA cells, and upon receiving such

cells, echo their contents to the corresponding TCP stream.

If the exit node does not support optimistic data (i.e. its version number is before 0.2.3.1-alpha), then the OP MUST wait for a RELAY_CONNECTED cell before sending any data. If the exit node supports optimistic data (i.e. its version number is 0.2.3.1-alpha or later), then the OP MAY send RELAY_DATA cells immediately after sending the RELAY_BEGIN cell (and before receiving either a RELAY_CONNECTED or RELAY_END cell).

RELAY_DATA cells sent to unrecognized streams are dropped. If the exit node supports optimistic data, then RELAY_DATA cells it receives on streams which have seen RELAY_BEGIN but have not yet been replied to with a RELAY_CONNECTED or RELAY_END are queued. If the stream creation succeeds with a RELAY_CONNECTED, the queue is processed immediately afterwards; if the stream creation fails with a RELAY_END, the contents of the queue are deleted.

Relay RELAY_DROP cells are long-range dummies; upon receiving such a cell, the OR or OP must drop it.

6.2.1. Opening a directory stream

If a Tor relay is a directory server, it should respond to a RELAY_BEGIN_DIR cell as if it had received a BEGIN cell requesting a connection to its directory port. RELAY_BEGIN_DIR cells ignore exit policy, since the stream is local to the Tor process.

If the Tor relay is not running a directory service, it should respond with a REASON_NOTDIRECTORY RELAY_END cell.

Clients MUST generate an all-zero payload for RELAY_BEGIN_DIR cells, and relays MUST ignore the payload.

[RELAY_BEGIN_DIR was not supported before Tor 0.1.2.2-alpha; clients SHOULD NOT send it to routers running earlier versions of Tor.]

6.3. Closing streams

When an anonymized TCP connection is closed, or an edge node encounters error on any stream, it sends a 'RELAY_END' cell along the circuit (if possible) and closes the TCP connection immediately. If an edge node receives a 'RELAY_END' cell for any stream, it closes the TCP connection completely, and sends nothing more along the circuit for that stream.

The payload of a RELAY_END cell begins with a single 'reason' byte to describe why the stream is closing, plus optional data (depending on the reason.) The values are:

```
    1 -- REASON_MISC           (catch-all for unlisted reasons)
    2 -- REASON_RESOLVEFAILED  (couldn't look up hostname)
    3 -- REASON_CONNECTREFUSED (remote host refused connection) [*]
    4 -- REASON_EXITPOLICY     (OR refuses to connect to host or port)
    5 -- REASON_DESTROY        (Circuit is being destroyed)
    6 -- REASON_DONE           (Anonymized TCP connection was closed)
    7 -- REASON_TIMEOUT        (Connection timed out, or OR timed out
                                while connecting)
    8 -- REASON_NOROUTE        (Routing error while attempting to
                                contact destination)
    9 -- REASON_HIBERNATING    (OR is temporarily hibernating)
   10 -- REASON_INTERNAL       (Internal error at the OR)
   11 -- REASON_RESOURCELIMIT  (OR has no resources to fulfill request)
   12 -- REASON_CONNRESET      (Connection was unexpectedly reset)
   13 -- REASON_TORPROTOCOL    (Sent when closing connection because of
                                Tor protocol violations.)
   14 -- REASON_NOTDIRECTORY   (Client sent RELAY_BEGIN_DIR to a
                                non-directory relay.)
```

(With REASON_EXITPOLICY, the 4-byte IPv4 address or 16-byte IPv6 address forms the optional data, along with a 4-byte TTL; no other reason currently has extra data.)

OPs and ORs MUST accept reasons not on the above list, since future versions of Tor may provide more fine-grained reasons.

Tors SHOULD NOT send any reason except REASON_MISC for a stream that they have originated.

[*] Older versions of Tor also send this reason when connections are reset.

--- [The rest of this section describes unimplemented functionality.]

Because TCP connections can be half-open, we follow an equivalent to TCP's FIN/FIN-ACK/ACK protocol to close streams.

An exit connection can have a TCP stream in one of three states: 'OPEN', 'DONE_PACKAGING', and 'DONE_DELIVERING'. For the purposes of modeling transitions, we treat 'CLOSED' as a fourth state, although connections in this state are not, in fact, tracked by the onion router.

A stream begins in the 'OPEN' state. Upon receiving a 'FIN' from the corresponding TCP connection, the edge node sends a 'RELAY_FIN' cell along the circuit and changes its state to 'DONE_PACKAGING'. Upon receiving a 'RELAY_FIN' cell, an edge node sends a 'FIN' to the corresponding TCP connection (e.g., by calling shutdown(SHUT_WR)) and changing its state to 'DONE_DELIVERING'.

When a stream in already in 'DONE_DELIVERING' receives a 'FIN', it also sends a 'RELAY_FIN' along the circuit, and changes its state to 'CLOSED'. When a stream already in 'DONE_PACKAGING' receives a 'RELAY_FIN' cell, it sends a 'FIN' and changes its state to 'CLOSED'.

If an edge node encounters an error on any stream, it sends a
'RELAY_END' cell (if possible) and closes the stream immediately.

6.4. Remote hostname lookup

To find the address associated with a hostname, the OP sends a
RELAY_RESOLVE cell containing the hostname to be resolved with a NUL
terminating byte. (For a reverse lookup, the OP sends a RELAY_RESOLVE
cell containing an in-addr.arpa address.) The OR replies with a
RELAY_RESOLVED cell containing any number of answers. Each answer is
of the form:

```
    Type   (1 octet)
    Length (1 octet)
    Value  (variable-width)
    TTL    (4 octets)
```

"Length" is the length of the Value field.
"Type" is one of:

```
   0x00 -- Hostname
   0x04 -- IPv4 address
   0x06 -- IPv6 address
   0xF0 -- Error, transient
   0xF1 -- Error, nontransient
```

If any answer has a type of 'Error', then no other answer may be given.

For backward compatibility, if there are any IPv4 answers, one of those
must be given as the first answer.

The RELAY_RESOLVE cell must use a nonzero, distinct streamID; the
corresponding RELAY_RESOLVED cell must use the same streamID. No stream
is actually created by the OR when resolving the name.

7. Flow control

7.1. Link throttling

Each client or relay should do appropriate bandwidth throttling to
keep its user happy.

Communicants rely on TCP's default flow control to push back when they
stop reading.

The mainline Tor implementation uses token buckets (one for reads,
one for writes) for the rate limiting.

Since 0.2.0.x, Tor has let the user specify an additional pair of
token buckets for "relayed" traffic, so people can deploy a Tor relay
with strict rate limiting, but also use the same Tor as a client. To
avoid partitioning concerns we combine both classes of traffic over a
given OR connection, and keep track of the last time we read or wrote
a high-priority (non-relayed) cell. If it's been less than N seconds
(currently N=30), we give the whole connection high priority, else we
give the whole connection low priority. We also give low priority
to reads and writes for connections that are serving directory
information. See proposal 111 for details.

7.2. Link padding

Link padding can be created by sending PADDING or VPADDING cells
along the connection; relay cells of type "DROP" can be used for
long-range padding. The contents of a PADDING, VPADDING, or DROP
cell SHOULD be chosen randomly, and MUST be ignored.

Currently nodes are not required to do any sort of link padding or
dummy traffic. Because strong attacks exist even with link padding,
and because link padding greatly increases the bandwidth requirements
for running a node, we plan to leave out link padding until this
tradeoff is better understood.

7.3. Circuit-level flow control

To control a circuit's bandwidth usage, each OR keeps track of two
'windows', consisting of how many RELAY_DATA cells it is allowed to
originate (package for transmission), and how many RELAY_DATA cells
it is willing to consume (receive for local streams). These limits
do not apply to cells that the OR receives from one host and relays
to another.

Each 'window' value is initially set based on the consensus parameter
'circwindow' in the directory (see dir-spec.txt), or to 1000 data cells
if no 'circwindow' value is given,
in each direction (cells that are not data cells do not affect
the window). When an OR is willing to deliver more cells, it sends a
RELAY_SENDME cell towards the OP, with Stream ID zero. When an OR
receives a RELAY_SENDME cell with stream ID zero, it increments its
packaging window.

Each of these cells increments the corresponding window by 100.

The OP behaves identically, except that it must track a packaging
window and a delivery window for every OR in the circuit.

An OR or OP sends cells to increment its delivery window when the
corresponding window value falls under some threshold (900).

If a packaging window reaches 0, the OR or OP stops reading from
TCP connections for all streams on the corresponding circuit, and
sends no more RELAY_DATA cells until receiving a RELAY_SENDME cell.
[this stuff is badly worded; copy in the tor-design section -RD]

7.4. Stream-level flow control

Edge nodes use RELAY_SENDME cells to implement end-to-end flow
control for individual connections across circuits. Similarly to

circuit-level flow control, edge nodes begin with a window of cells (500) per stream, and increment the window by a fixed value (50) upon receiving a RELAY_SENDME cell. Edge nodes initiate RELAY_SENDME cells when both a) the window is <= 450, and b) there are less than ten cell payloads remaining to be flushed at that edge.

8. Handling resource exhaustion

8.1. Memory exhaustion.

If RAM becomes low, an OR should begin destroying circuits until more memory is free again. We recommend the following algorithm:

- Set a threshold amount of RAM to recover at 10% of the total RAM.
- Sort the circuits by their 'staleness', defined as the age of the oldest data queued on the circuit. This data can be:
    * Bytes that are waiting to flush to or from a stream on that circuit.
    * Bytes that are waiting to flush from a connection created with BEGIN_DIR.
    * Cells that are waiting to flush or be processed.
- While we have not yet recovered enough RAM:
    * Free all memory held by the most stale circuit, and send DESTROY cells in both directions on that circuit. Count the amount of memory we recovered towards the total.

9. Subprotocol versioning

This section specifies the Tor subprotocol versioning. They are broken down into different types with their current version numbers. Any new version number should be added to this section.

The dir-spec.txt details how those versions are encoded. See the "proto"/"pr" line in a descriptor and the "recommended-relay-protocols", "required-relay-protocols", "recommended-client-protocols" and "required-client-protocols" lines in the vote/consensus format.

Here are the rules a relay and client should follow when encountering a protocol list in the consensus:

- When a relay lacks a protocol listed in recommended-relay-protocols, it should warn its operator that the relay is obsolete.
- When a relay lacks a protocol listed in required-relay-protocols, it must not attempt to join the network.
- When a client lacks a protocol listed in recommended-client-protocols, it should warn the user that the client is obsolete.
- When a client lacks a protocol listed in required-client-protocols, it must not connect to the network. This implements a "safe forward shutdown" mechanism for zombie clients.
- If a client or relay has a cached consensus telling it that a given protocol is required, and it does not implement that protocol, it SHOULD NOT try to fetch a newer consensus.

Starting in version 0.2.9.4-alpha, the initial required protocols for clients that we will Recommend and Require are:

```
Cons=1-2 Desc=1-2 DirCache=1 HSDir=2 HSIntro=3 HSRend=1 Link=4
LinkAuth=1 Microdesc=1-2 Relay=2
```

For relays we will Require:

```
Cons=1 Desc=1 DirCache=1 HSDir=2 HSIntro=3 HSRend=1 Link=3-4
LinkAuth=1 Microdesc=1 Relay=1-2
```

For relays, we will additionally Recommend all protocols which we recommend for clients.

9.1. "Link"

The "link" protocols are those used by clients and relays to initiate and receive OR connections and to handle cells on OR connections. The "link" protocol versions correspond 1:1 to those versions.

Two Tor instances can make a connection to each other only if they have at least one link protocol in common.

The current "link" versions are: "1" through "4". See section 4.1 for more information. All current Tor versions support "1-3"; version from 0.2.4.11-alpha and on support "1-4". Eventually we will drop "1" and "2".

9.2. "LinkAuth"

LinkAuth protocols correspond to varieties of Authenticate cells used for the v3+ link protocools.

The current version is "1".

"2" is unused, and reserved by proposal 244.

"3" is the ed25519 link handshake of proposal 220.

9.3. "Relay"

The "relay" protocols are those used to handle CREATE cells, and those that handle the various RELAY cell types received after a CREATE cell. (Except, relay cells used to manage introduction and rendezvous points are managed with the "HSIntro" and "HSRend" protocols respectively.)

Current versions are:

"1" -- supports the TAP key exchange, with all features in Tor 0.2.3. Support for CREATE and CREATED and CREATE_FAST and CREATED_FAST and EXTEND and EXTENDED.

"2" -- supports the ntor key exchange, and all features in Tor 0.2.4.19. Includes support for CREATE2 and CREATED2 and EXTEND2 and EXTENDED2.

9.4. "HSIntro"

The "HSIntro" protocol handles introduction points.

"3" -- supports authentication as of proposal 121 in Tor 0.2.1.6-alpha.

"4" -- support ed25519 authentication keys which is defined by the HS v3 protocol as part of proposal 224 in Tor 0.3.0.4-alpha.

9.5. "HSRend"

The "HSRend" protocol handles rendezvous points.

"1" -- supports all features in Tor 0.0.6.

"2" -- supports RENDEZVOUS2 cells of arbitrary length as long as they have 20 bytes of cookie in Tor 0.2.9.1-alpha.

9.6. "HSDir"

The "HSDir" protocols are the set of hidden service document types that can be uploaded to, understood by, and downloaded from a tor relay, and the set of URLs available to fetch them.

"1" -- supports all features in Tor 0.2.0.10-alpha.

"2" -- support ed25519 blinded keys request which is defined by the HS v3 protocol as part of proposal 224 in Tor 0.3.0.4-alpha.

9.7. "DirCache"

The "DirCache" protocols are the set of documents available for download from a directory cache via BEGIN_DIR, and the set of URLs available to fetch them. (This excludes URLs for hidden service objects.)

"1" -- supports all features in Tor 0.2.4.19.

9.8. "Desc"

Describes features present or absent in descriptors.

Most features in descriptors don't require a "Desc" update -- only those that need to someday be required. For example, someday clients will need to understand ed25519 identities.

"1" -- supports all features in Tor 0.2.4.19.

"2" -- cross-signing with onion-keys, signing with ed25519 identities.

9.9. "Microdesc"

Describes features present or absent in microdescriptors.

Most features in descriptors don't require a "MicroDesc" update -- only those that need to someday be required. These correspond more or less with consensus methods.

"1" -- consensus methods 9 through 20.

"2" -- consensus method 21 (adds ed25519 keys to microdescs).

9.10. "Cons"

Describes features present or absent in consensus documents.

Most features in consensus documents don't require a "Cons" update -- only those that need to someday be required.

These correspond more or less with consensus methods.

"1" -- consensus methods 9 through 20.

"2" -- consensus method 21 (adds ed25519 keys to microdescs).