> The reason not to cache an address is that the exit might have lied about the actual address of the host, or might have given us a unique address to identify us in the future.

[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.]

Transmitting data

Once a connection has been established, the client and exit node package stream data in RELAY_DATA message, and upon receiving such messages, echo their contents to the corresponding TCP stream.

The client MAY send RELAY_DATA messages immediately after sending the RELAY_BEGIN message (and before receiving either a RELAY_CONNECTED or RELAY_END message).

> In some contexts, messages sent before receiving a RELAY_CONNECTED message are called “optimistic data”.

When an exit receives RELAY_DATA messages it receives on streams which have seen RELAY_BEGIN but have not yet been replied to with a RELAY_CONNECTED or RELAY_END, those messages are 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_DATA messages sent to closed streams are dropped.

RELAY_DATA messages sent to unrecognized streams are an error, and cause the circuit to close.

Relay RELAY_DROP messages are long-range dummies; upon receiving such a message, the relay or client must drop it.

Opening a directory stream

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

Tor specifications

These specifications describe how Tor works. They try to present Tor’s protocols in sufficient detail to allow the reader to implement a compatible implementation of Tor without ever having to read the Tor source code.

They were once a separate set of text files, but in late 2023 we migrated to use `mdbook`. We’re in the process of updating these documents to improve their quality.

This is a living document: we are always changing and improving them in order to make them easier and more accurate, and to improve the quality of the Tor protocols. They are maintained as a set of documents in a gitlab repository; you can use that repository to see their history.

Additionally, the proposals directory holds our design proposals. These include historical documents that have now been merged into the main specifications, and new proposals that are still under discussion. Of particular interest are the `FINISHED` Tor proposals: They are the ones that have been implemented, but not yet merged into these documents.

Getting started

There’s a lot of material here, and it’s not always as well organized as we like. We have broken it into a few major sections.

For a table of contents, click on the menu icon to the top-left of your browser scene. You should probably start by reading the core tor protocol specification, which describes how our protocol works. After that, you should be able to jump around to the topics that interest you most. The introduction of each top-level chapter should provide an introduction.

A short introduction to Tor

Basic functionality

Tor is a distributed overlay network designed to anonymize low-latency TCP-based applications such as web browsing, secure shell, and instant messaging. The network is built of a number of servers, called **relays** (also called “onion routers” or “ORs” in some older documentation).

To connect to the network, a client needs to download an up-to-date signed directory of the relays on the network. These directory documents are generated and signed by a set of semi-trusted **directory authority** servers, and are cached by the relays themselves. (If a client does not yet have a directory, it finds a cache by looking at a list of stable cache locations, distributed along with its source code.)

> For more information on the directory subsystem, see the directory protocol specification.

After the client knows the relays on the network, it can pick a relay and open a **channel** to one of these relays. A channel is an encrypted reliable non-anonymous transport between a client and a relay or a relay and a relay, used to transmit messages called **cells**. (Under the hood, a channel is just a TLS connection over TCP, with a specified encoding for cells.)

To anonymize its traffic, a client chooses a **path**—a sequence of relays on the network— and opens a channel to the first relay on the path (if it does not already have a channel open to that relay). The client then uses that channel to build a multi-hop cryptographic structure called a **circuit**. A circuit is built over a sequence of relays (typically three). Every relay in the circuit knows its precessor and successor, but no other relays in the circuit. Many circuits can be multiplexed over a single channel.

```
      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 message, the exit node first checks whether it happens to be the first node in the circuit (i.e. someone is trying to create a one-hop circuit). It does so by inspecting the accompanying channel to determine whether the channel initiator has authenticated itself, and whether its fingerprint is part of the current consensus. (See “Negotiating and initializing channels”)

Any attempts to create a one-hop circuit using a RELAY_BEGIN message SHOULD be declined by sending an appropriate DESTROY cell with a protocol violation as its reason.

Afterwards, 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 nodes replies with a RELAY_END message. (See “Closing streams”) Otherwise, the exit node replies with a RELAY_CONNECTED message, whose body is 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]
```

Implementations MUST accept either of these formats, and MUST also accept an empty RELAY_CONNECTED message body.

Implementations MAY ignore the address value, and MAY choose not to cache it. If an implementation chooses to cache the address, it SHOULD NOT reuse that address with any other circuit.

Opening streams and transmitting data

Opening a new stream: The begin/connected handshake

To open a new anonymized TCP connection, the client 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 message with a body encoding the address and port of the destination host. The body format is:

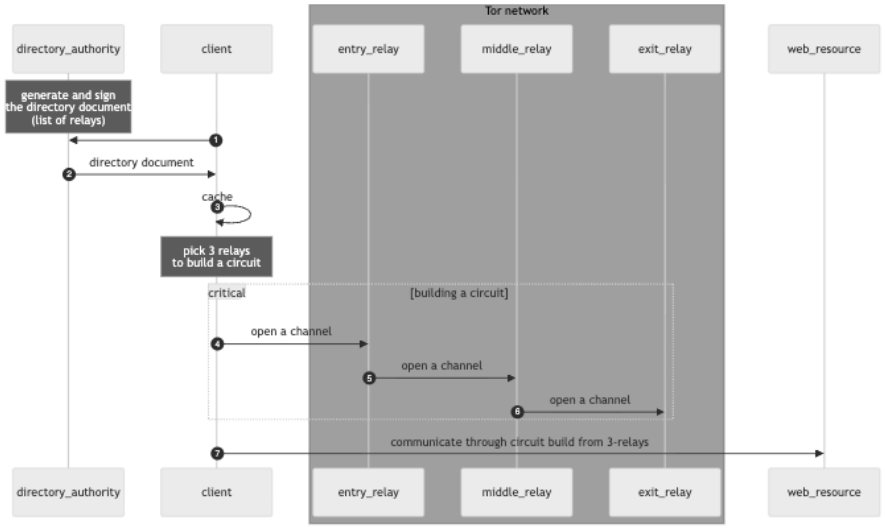
```
ADDRPORT [nul-terminated string]
FLAGS    [4 bytes, optional]

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 ADDRPORT string SHOULD be sent in lower case, to avoid fingerprinting. Implementations MUST accept strings in any case.

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 integers in Tor are big-endian, 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.)

If FLAGS is absent, its value is 0. Whenever 0 would be sent for FLAGS, FLAGS is omitted from the message body.



> For more information on how paths are selected, see the path specification. The first hop on a path, also called a **guard node**, has complicated rules for its selection; for more on those, see the guard specification.

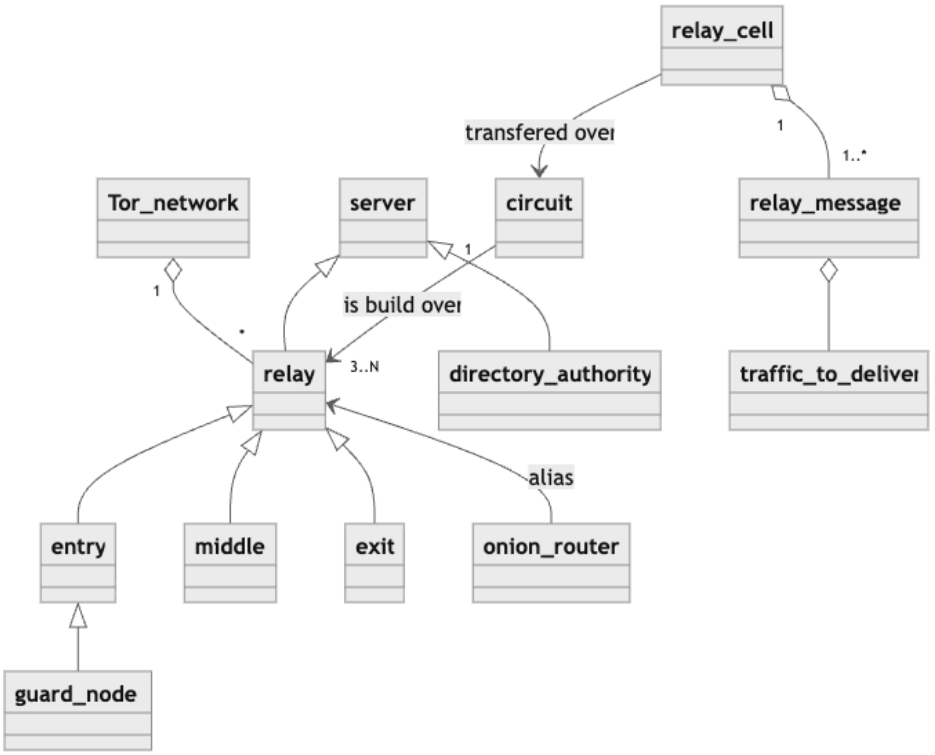
Once a circuit exists, the client can use it to exchange fixed-length **relay cells** with any relay on the circuit. These relay cells are wrapped in multiple layers of encryption: as part of building the circuit, the client negotiates a separate set of symmetric keys with each relay on the circuit. Each relay removes (or adds) a single layer of encryption for each relay cell before passing it on.

A client uses these relay cells to exchange **relay messages** with relays on a circuit. These “relay messages” in turn are used to actually deliver traffic over the network. In the simplest use case, the client sends a `BEGIN` message to tell the last relay on the circuit (called the **exit node**) to create a new session, or **stream**, and associate that stream with a new TCP connection to a target host. The exit node replies with a `CONNECTED` message to say that the TCP connection has succeeded. Then the client and the exit exchange `DATA` messages to represent the contents of the anonymized stream.

> Note that as of 2023, the specifications do not perfectly distinguish between relay cells and relay messages. This is because, until recently, there was a 1-to-1 relationship between the two: every relay cell held a single relay

message. As proposal 340 is implemented, we will revise the specifications for improved clarify on this point.

Other kinds of relay messages can be used for more advanced functionality.



Using a system called **conflux** a client can build multiple circuits to the *same* exit node, and associate those circuits within a **conflux set**. Once this is done, relay messages can be sent over *either* circuit in the set, depending on capacity and performance.

hash function used here is SHA-1. For onion service circuits, the hash function is SHA3-256.

When ENCRYPTING a relay cell, an implementation does the following:

```
# Encode the cell in binary (recognized and digest set to zero)
tmp = cmd + [0, 0] + stream_id + [0, 0, 0, 0] + length + data +
padding

# Update the hash state with the encoded data
hash_state = hash_update(hash_state, tmp)
digest = hash_calculate(hash_state)

# The encoded data is the same as above with the digest field not
being
# zero anymore
encoded = cmd + [0, 0] + stream_id + digest[0..4] + length + data +
          padding

# Now we can encrypt the cell by adding the onion layers ...
```

When DECRYPTING a relay cell, an implementation does the following:

```
decrypted = decrypt(cell)

# Replace the digest field in decrypted by zeros
tmp = decrypted[0..5] + [0, 0, 0, 0] + decrypted[9..]

# Update the digest field with the decrypted data and its digest field
# set to zero
hash_state = hash_update(hash_state, tmp)
digest = hash_calculate(hash_state)

if digest[0..4] == decrypted[5..9]
  # The cell has been fully decrypted ...
```

The caveat itself is that only the binary data with the digest bytes set to zero are being taken into account when calculating the running digest. The final plain-text cells (with the digest field set to its actual value) are not taken into the running digest.

cell, not only those up to “Len”. If the digest is correct, the cell is considered “recognized” for the purposes of decryption (see Routing relay cells).

(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 messages pertaining to the same tunneled stream have the same stream ID. StreamIDs are chosen arbitrarily by the client. No stream may have a StreamID of zero. Rather, relay messages that affect the entire circuit rather than a particular stream use a StreamID of zero – they are marked in the table above as “**C**” ([control“) style cells. (Sendme cells are marked as “sometimes control” because they can include a StreamID or not depending on their purpose – see Flow control.)

The ‘Length’ field of a relay cell contains the number of bytes in the relay cell’s body which contain the body of the message. The remainder of the unencrypted relay cell’s body is padded with padding bytes. Implementations handle padding bytes of unencrypted relay cells as they do padding bytes for other cell types; see Cell Packet format.

The ‘Padding’ field is used to make relay cell contents unpredictable, to avoid certain attacks (see proposal 289 for rationale). Implementations SHOULD fill this field with four zero-valued bytes, followed by as many random bytes as will fit. (If there are fewer than 4 bytes for padding, then they should all be filled with zero.

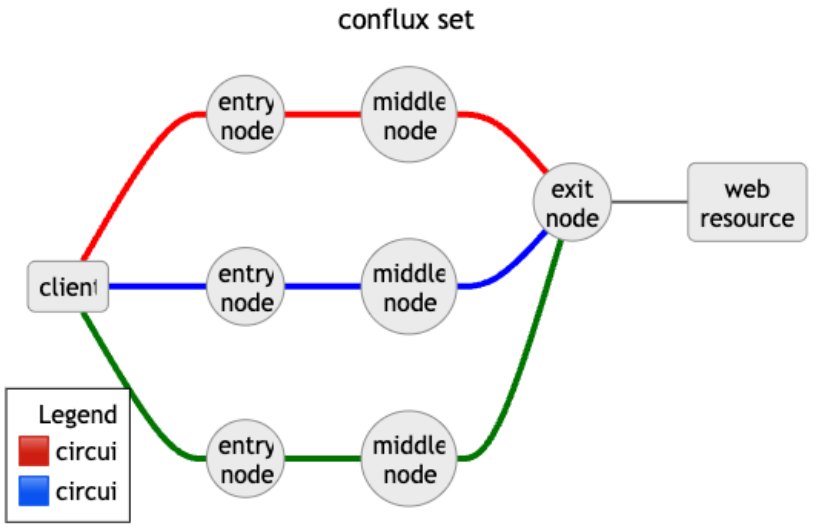
Implementations MUST NOT rely on the contents of the ‘Padding’ field.

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.

Calculating the ‘Digest’ field

The ‘Digest’ field itself serves the purpose to check if a cell has been fully decrypted, that is, all onion layers have been removed. Having a single field, namely ‘Recognized’ is not sufficient, as outlined above.

In this section, we assume an incrementally updated hash function, where `hash_calculate(state)` computes the current digest, and `hash_update(state,M)` adjusts the hash function’s state by adding `M` to its input. For ordinary circuits, the



For more on conflux, which has been integrated into the C tor implementation, but not yet (as of 2023) into this document, see proposal 329.

Advanced topics: Onion services and responder anonymity

In addition to *initiating* anonymous communications, clients can also arrange to *receive* communications without revealing their identity or location. This is called **responder anonymity**, and the mechanism Tor uses to achieve it is called **onion services** (or “hidden services” or “rendezvous services” in some older documentation).

For the details on onion services, see the Tor Rendezvous Specification.

Advanced topics: Censorship resistence

In some places, Tor is censored. Typically, censors do this by blocking connections to the addresses of the known Tor relays, and by blocking traffic that resembles Tor.

To resist this censorship, some Tor relays, called **bridges**, are unlisted in the public directory: their addresses are distributed by other means (To distinguish ordinary published relays from bridges, we sometimes call them **public relays**.)

Additionally, Tor clients and bridges can use extension programs, called **pluggable transports**, that obfuscate their traffic to make it harder to detect.

Command	Identifier	Type	Description
			Rendezvous request (to rendezvous point)
37	RENDEZVOUS2	**B**, **C**	Rendezvous request (to client)
38	INTRO_ESTABLISHED	**B**, **C**	Acknowledge ESTABLISH_INTRO
39	RENDEZVOUS_ESTABLISHED	**B**, **C**	Acknowledge ESTABLISH_RENDEZVOUS
40	INTRODUCE_ACK	**B**, **C**	Acknowledge INTRODUCE1
	Circuit padding		
41	PADDING_NEGOTIATE	**F**, **C**	Negotiate circuit padding
42	PADDING_NEGOTIATED	**B**, **C**	Negotiate circuit padding
	Flow control		
43	XOFF	**F**/**B**	Stream-level flow control
44	XON	**F**/**B**	Stream-level flow control

- **F** (Forward): Must only be sent by the originator of the circuit.
- **B** (Backward): Must only be sent by other nodes in the circuit back towards the originator.
- **F**/**B** (Forward or backward): May be sent in either direction.
- **C**: (Control) must have a zero-valued stream ID. (Other commands must have a nonzero stream ID.)

The 'recognized' field is used as a simple indication that the cell is still encrypted. It is an optimization to avoid calculating expensive digests for every cell. When sending cells, the unencrypted 'recognized' MUST be set to zero.

When receiving and decrypting cells the 'recognized' will always be zero if we're the endpoint that the cell is destined for. For cells that we should relay, the 'recognized' field will usually be nonzero, but will accidentally be zero with P=2^-16.

When handling a relay cell, if the 'recognized' in field in a decrypted relay cell is 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 Setting circuit keys), and including this relay cell's entire body (taken with the digest field set to zero). Note that these digests *do* include the padding bytes at the end of the

Command	Identifier	Type	Description
5	SENDME	**F/B, C?**	Acknowledge traffic
6	EXTEND	**F, C**	Extend a circuit with TAP (obsolete)
7	EXTENDED	**B, C**	Finish extending a circuit with TAP (obsolete)
8	TRUNCATE	**F, C**	Remove nodes from a circuit (unused)
9	TRUNCATED	**B, C**	Report circuit truncation (unused)
10	DROP	**F/B, C**	Long-range padding
11	RESOLVE	**F**	Hostname lookup
12	RESOLVED	**B**	Hostname lookup reply
13	BEGIN_DIR	**F**	Open stream to directory cache
14	EXTEND2	**F, C**	Extend a circuit
15	EXTENDED2	**B, C**	Finish extending a circuit
16..18	Reserved		For UDP; see prop339.
	Conflux		
19	CONFLUX_LINK	**F, C**	Link circuits into a bundle
20	CONFLUX_LINKED	**B, C**	Acknowledge link request
21	CONFLUX_LINKED_ACK	**F, C**	Acknowledge CONFLUX_LINKED message (for timing)
22	CONFLUX_SWITCH	**F/B, C**	Switch between circuits in a bundle
	Onion services		
32	ESTABLISH_INTRO	**F, C**	Create introduction point
33	ESTABLISH_RENDEZVOUS	**F, C**	Create rendezvous point
34	INTRODUCE1	**F, C**	Introduction request (to intro point)
35	INTRODUCE2	**B, C**	Introduction request (to service)
36	RENDEZVOUS1	**F, C**	

Notation and conventions

These conventions apply, at least in theory, to all of the specification documents unless stated otherwise.

> Remember, our specification documents were once a collection of separate text files, written separately and edited over the course of years.
>
> While we are trying (as of 2023) to edit them into consistency, you should be aware that these conventions are not now followed uniformly everywhere.

MUST, SHOULD, and so on

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.

Data lengths

Lengths are given as a number of 8-bit bytes.

> All bytes are 8 bits long. We sometimes call them “octets”; the terms as used here are interchangeable.

When referring to longer lengths, we use SI binary prefixes (as in “kibibytes”, “mebibytes”, and so on) to refer unambiguously to increments of 1024^X bytes.

> If you encounter a reference to “kilobytes”, “megabytes”, or so on, you cannot safely infer whether the author intended a decimal (1000^N) or binary (1024^N) interpretation. In these cases, it is better to revise the specifications.

Integer encoding

Multi-byte integers are encoded in big-endian (“network”) order.

> For example, 4660 (0x1234), when encoded as a two-byte integer, is the byte 0x12 followed by the byte 0x34. ([12 34])
>
> When encoded as a four-byte integer, it is the byte 0x00, the byte 0x00, the byte 0x12, and the byte 0x34. ([00 00 12 34]).

Textual formats

The Tor Protocols involve textual elements (for example, text files such as netdocs, command lines, and bridge lines).

Strings, including keywords (for example, netdoc Item keywords, network parameters, and router flags), are compared for equality as byte strings. Likewise, "lexical ordering" of strings is lexical ordering of byte strings.

> This means that these keywords are compared case-sensitively, without any kind of Unicode normalisation, etc.

Binary-as-text encodings

When we refer to "base64", "base32", or "base16", we mean the encodings described in RFC 4648, with the following notes:

- In base32, we never insert linefeeds in base32, and we omit trailing = padding characters.
- In base64, we *sometimes* omit trailing = padding characters, and we do not insert linefeeds unless explicitly noted.
- We do not insert any other whitespace, except as specifically noted.

Base 16 and base 32 are case-insensitive. Implementations should accept any cases, and should produce a single uniform case.

We sometimes refer to base16 as "hex" or "hexadecimal".

> Note that as of 2023, in some places, the specs are not always explicit about:
>
> - which base64 strings are multiline
> - which base32 strings and base16 strings should be generated in what case.
>
> This is something we should correct.

Notation

Operations on byte strings

- `A | B` represents the concatenation of two binary strings `A` and `B`.

Relay cells

Within a circuit, the client and the end node use the contents of relay cells 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 client.

End nodes that accept streams may be:

- exit relays (RELAY_BEGIN, anonymous),
- directory servers (RELAY_BEGIN_DIR, anonymous or non-anonymous),
- onion services (RELAY_BEGIN, anonymous via a rendezvous point).

The body of each unencrypted relay cell consists of an enveloped relay message, encoded as follows:

Field	Size
Relay command	1 byte
'Recognized'	2 bytes
StreamID	2 bytes
Digest	4 bytes
Length	2 bytes
Data	Length bytes
Padding	CELL_BODY_LEN - 11 - Length bytes

> TODO: When we implement prop340, we should clarify which parts of the above are about the relay cell, and which are the enveloped message.

The relay commands are:

Command	Identifier	Type	Description
	Core protocol		
1	BEGIN	**F**	Open a stream
2	DATA	**F/B**	Transmit data
3	END	**F/B**	Close a stream
4	CONNECTED	**B**	Stream has successfully opened

Application connections and stream management

This section describes how clients use relay messages to communicate with exit nodes, and how use this communication channel to send and receive application data.

Binary literals

When we write a series of one-byte hexadecimal literals in square brackets, it represents a multi-byte binary string.

> For example, `[6f 6e 69 6f 6e 20 72 6f 75 74 69 6e 67]` is a 13-byte sequence representing the unterminated ASCII string, `onion routing`.

Tor Protocol Specification

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. 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.

Handling RELAY_EARLY cells

A RELAY_EARLY cell is designed to limit the length any circuit can reach. When a relay receives a RELAY_EARLY cell, and the next node in the circuit is speaking v2 of the link protocol or later, the relay 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/EXTEND2 message 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/EXTEND2 cell not received in a RELAY_EARLY cell.]

The relay then decides whether it recognizes the relay cell, by inspecting the cell as described in Relay cells. If the relay 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 relay at the end of the circuit encounters an unrecognized relay cell, an error has occurred: the relay sends a DESTROY cell to tear down the circuit.

For more information, see Application connections and stream management.

Backward Direction

The backward direction is the opposite direction from CREATE/CREATE2 cells.

Relaying Backward at Onion Routers

When a backward relay cell is received by a relay, it encrypts the cell's body with the stream cipher, as follows:

```
'Backward' relay cell:
   Use Kb as key; encrypt.
```

Routing to the Origin

When a relay cell arrives at a client, the client decrypts the cell's body with the stream cipher as follows:

```
Client receives relay cell from node 1:
   For I=1...N, where N is the final node on the circuit:
       Decrypt with Kb_I.
       If the cell is recognized (see [1]), then:
           The sending node is I.
           Stop and process the cell.
```

[1]: "Relay cells"

Preliminaries

Notation and encoding

```
   KP -- a public key for an asymmetric cipher.
   KS -- a private key for an asymmetric cipher.
   K  -- a key for a symmetric cipher.
   N  -- a "nonce", a random value, usually deterministically chosen
         from other inputs using hashing.
```

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.
   KP_ENC_LEN -- the length of a public-key encrypted message, in
bytes.
   KP_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 KP_ENC_LEN-
KP_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).
```

Message lengths

Some message lengths are fixed in the Tor protocol. We give them here. Some of these message lengths depend on the version of the Tor link protocol in use: for these, the link protocol is denoted in this table with v.

Name	Length in bytes	Meaning
`CELL_BODY_LEN`	509	The body length for a fixed-length cell.
	2	The length of a circuit ID

Name	Length in bytes	Meaning
`CIRCID_LEN(v)`, v < 4		
`CIRCID_LEN(v)`, v ≥ 4	4	
`CELL_LEN(v)`, v < 4	512	The length of a fixed-length cell.
`CELL_LEN(v)`, v ≥ 4	514	

Note that for all v, `CELL_LEN(v) = 1 + CIRCID_LEN(v) + CELL_BODY_LEN`.

> Formerly `CELL_BODY_LEN` was called sometimes called `PAYLOAD_LEN`.

Ciphers

These are the ciphers we use *unless otherwise specified*. Several of them are deprecated for new use.

For a stream cipher, unless otherwise specified, we use 128-bit AES in counter mode, with an IV of all 0 bytes. (We also require AES256.)

For a public-key cipher, unless otherwise specified, 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)

We also use the Curve25519 group and the Ed25519 signature format in several places.

For Diffie-Hellman, unless otherwise specified, 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.

Routing relay cells

Circuit ID Checks

When a node wants to send a RELAY or RELAY_EARLY cell, it checks the cell's circID and determines whether the corresponding circuit along that connection is still open. If not, the node drops the cell.

When a node 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 node drops the cell.

> Here and elsewhere, we refer to RELAY and RELAY_EARLY cells collectively as "relay cells".

Forward Direction

The forward direction is the direction that CREATE/CREATE2 cells are sent.

Routing from the Origin

When a relay cell is sent from a client, the client encrypts the cell's body with the stream cipher as follows:

```
Client sends relay cell:
   For I=N...1, where N is the destination node:
      Encrypt with Kf_I.
   Transmit the encrypted cell to node 1.
```

Relaying Forward at Onion Routers

When a forward relay cell is received by a relay, it decrypts the cell's body with the stream cipher, as follows:

```
'Forward' relay cell:
   Use Kf as key; decrypt.
```

```
     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 -- CHANNEL_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)
```

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]

KEY_LEN=16. DH_LEN=128; DH_SEC_LEN=40. KP_ENC_LEN=128; KP_PAD_LEN=42.

All "random" values MUST be generated with a cryptographically strong pseudorandom number generator seeded from a strong entropy source, unless otherwise noted. All "random" values MUST selected uniformly at random from the universe of possible values, unless otherwise noted.

Cryptographic hash functions

Tor uses the cryptographic hash functions SHA-1, SHA-256, and SHA3-256.

> SHA-1 is vulnerable to various collision attacks, and should not be used anywhere new. Its existing applications are redundant with other hash functions, deprecated, or both.

We denote applications of these hash functions to some message M as:

- `SHA1(M)`
- `SHA256(M)`
- `SHA3_256(M)`.

We define constants to represent the lengths in bytes of the digests that these functions output:

```
SHA1_LEN = 20
SHA256_LEN = 32
SHA3_256_LEN = 32
```

Note that although the above terminology is preferred, many of our older specifications have not yet been converted to use it. In some places, we also use `H(M)` to mean “the digest of M”, and `DIGEST_LEN` or `HASH_LEN` to refer to the length of that digest. Unless otherwise specified, `H(M)` is computed using SHA-1.

Computing the digest of an RSA key

When `key` is an RSA public key, we use the notation `DER(key)` to denote the ASN.1 DER encoding of the key’s representation as a PKCS#1 RSAPublicKey object.

Some older text does not use yet this notation. When we refer to “the digest of an RSA public key”, unless otherwise specified, we mean a digest of `DER(key)`. The hash function should be specified explicitly.

[Note: If a relay receives a TRUNCATE message 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 message 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 a circuit, the nodes
   must report it as follows:
     * If possible, send a DESTROY cell to relays _away_ from the
client.
     * If possible, send *either* a DESTROY cell towards the client,
or
       a RELAY_TRUNCATED cell towards the client.
```

Current versions of Tor do not reuse truncated RELAY_TRUNCATED circuits: A client, upon receiving a RELAY_TRUNCATED, will send forward a DESTROY cell in order to entirely tear down the circuit. Because of this, we recommend that relays should send DESTROY towards the client, not RELAY_TRUNCATED.

```
   NOTE:
     In tor versions before 0.4.5.13, 0.4.6.11 and 0.4.7.9, relays
would
     handle an inbound DESTROY by sending the client a RELAY_TRUNCATED
     message.  Beginning with those versions, relays now propagate
     DESTROY cells in either direction, in order to tell every
     intermediary relays to stop queuing data on the circuit.  The
earlier
     behavior created queuing pressure on the intermediary relays.
```

The body of a DESTROY cell or RELAY_TRUNCATED message contains a single octet, describing the reason that the circuit was closed. RELAY_TRUNCATED message, and DESTROY cells sent _towards the client, should contain the actual reason from the list of error codes below. Reasons in DESTROY cell SHOULD NOT be propagated downward or upward, due to potential side channel risk: A relay receiving a DESTROY command should use the DESTROYED reason for its next cell. A client should always use the NONE reason for its own DESTROY cells.

The error codes are:

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.

Relays SHOULD also tear down circuits which attempt to create:

- streams with RELAY_BEGIN, or
- rendezvous points with ESTABLISH_RENDEZVOUS, ending at the first hop. Letting Tor be used as a single hop proxy makes exit and rendezvous nodes a more attractive target for compromise.

Relays MAY use multiple methods to check if they are the first hop:

```
    * If a relay sees a circuit created with CREATE_FAST, the relay is
 sure to be
      the first hop of a circuit.
    * If a relay is the responder, and the initiator:
      * did not authenticate the link, or
      * authenticated with a key that is not in the consensus,
      then the relay is probably the first hop of a circuit (or the
 second hop of
      a circuit via a bridge relay).

    Circuits may be torn down either completely or hop-by-hop.
```

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

Upon receiving an outgoing DESTROY cell, a relay 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 relay in the circuit. If the node is the end of the circuit, then it tears down any associated edge connections (see Calculating the 'Digest' field).

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

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

Relay keys and identities

Every Tor relay has multiple public/private keypairs, with different lifetimes and purposes. We explain them here.

Each key here has an English name (like "Ed25519 identity key") and an unambiguous identifier (like `KP_relayid_ed`).

In an identifier, a `KP_` prefix denotes a public key, and a `KS_` prefix denotes the corresponding secret key.

> For historical reasons or reasons of space, you will sometimes encounter multiple English names for the same key, or shortened versions of that name. The identifier for a key, however, should always be unique and unambiguous.

For security reasons, **all keys MUST be distinct**: the same key or keypair should never be used for separate roles within the Tor protocol suite, unless specifically stated. For example, a relay's identity key `KP_relayid_ed` MUST NOT also be used as its medium-term signing key `KP_relaysign_ed`.

Identity keys

An **identity key** is a long-lived key that uniquely identifies a relay. Two relays with the same set of identity keys are considered to be the same; any relay that changes its identity key is considered to have become a different relay.

An identity keypair's lifetime is the same as the lifetime of the relay.

Two identity keys are currently defined:

- `KP_relayid_ed`, `KS_relayid_ed`: An "ed25519 identity key", also sometimes called a "master identity key".

  This is an Ed25519 key. This key never expires. It is used for only one purpose: signing the `KP_relaysign_ed` key, which is used to sign other important keys and objects.

- `KP_relayid_rsa`, `KS_relayid_rsa`: A *legacy* "RSA identity key".

This is an RSA key. It never expires. It is always 1024 bits long, and (as discussed above) its exponent must be 65537. It is used to sign directory documents and certificates.

Note that because the legacy RSA identity key is so short, it should not be assumed secure against an attacker. It exists for legacy purposes only. When authenticating a relay, a failure to prove an expected RSA identity is sufficient evidence of a *failure* to authenticate, but a successful proof of an RSA identity is not sufficient to establish a relay's identity. Parties SHOULD NOT use the RSA identity on its own.

We write `KP_relayid` to refer to a key which is either `KP_relayid_rsa` or `KP_relayid_ed`.

Online signing keys

Since Tor's design tries to support keeping the high-value Ed25519 relay identity key offline, we need a corresponding key that can be used for online signing:

- `KP_relaysign_ed`, `KS_relaysign_ed`: A medium-term Ed25519 "signing" key. This key is signed by the identity key `KP_relayid_ed`, and must be kept online. A new one should be generated periodically. It signs nearly everything else, including directory objects, and certificates for other keys.

When this key is generated, it needs to be signed with the `KP_relayid_ed` key, producing a certificate of type `IDENTITY_V_SIGNING`. The `KP_relayid_ed` key is not used for anything else.

Circuit extension keys

Each relay has one or more **circuit extension keys** (also called "onion keys"). When creating or extending a circuit, a client uses this key to perform a one-way authenticated key exchange with the target relay. If the recipient does not have the correct private key, the handshake will fail.

Circuit extension keys have moderate lifetimes, on the order of weeks. They are published as part of the directory protocol, and relays SHOULD accept handshakes for a while after publishing any new key. (The exact durations for these are set via a set of network parameters.)

- The relay knows that the IP of the connection it's using is canonical because it was listed in the NETINFO cell.
- The IP matches the relay address in the consensus.

When an onion router receives an EXTEND2 relay message, it sends a CREATE2 cell to the next onion router, with the enclosed HLEN, HTYPE, and HDATA as its body. The initiating onion router chooses some circID not yet used on the connection between the two onion routers. (But see section "Choosing circuit IDs in create cells")

As special cases, if the EXTEND/EXTEND2 message includes a legacy identity, or identity fingerprint of all zeroes, or asks to extend back to the relay that sent the extend cell, the circuit will fail and be torn down.

Ed25519 identity keys are not required in EXTEND2 messages, so all zero keys SHOULD be accepted. If the extending relay knows the ed25519 key from the consensus, it SHOULD also check that key. (See EXTEND and EXTENDED message)

If an EXTEND2 message contains the ed25519 key of the relay that sent the EXTEND2 message, the circuit will fail and be torn down.

When an onion router receives a CREATE/CREATE2 cell, if it already has a circuit on the given connection with the given circID, it drops the cell. Otherwise, after receiving the CREATE/CREATE2 cell, it completes the specified handshake, and replies with a CREATED/CREATED2 cell.

Upon receiving a CREATED/CREATED2 cell, an onion router packs its body into an EXTENDED/EXTENDED2 relay message, and sends that message up the circuit. Upon receiving the EXTENDED/EXTENDED2 relay message, the client can retrieve the handshake material.

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

Canonical connections

It is possible for an attacker to launch a man-in-the-middle attack against a connection by telling relay Alice to extend to relay 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 a relay gets an extend request, it SHOULD use an existing relay connection if the ID matches, and ANY of the following conditions hold:

- The IP matches the requested IP.

There are two current kinds of circuit extension keys:

- `KP_ntor`, `KS_ntor`: A curve25519 key used for the `ntor` and `ntorv3` circuit extension handshakes.
- `KP_onion_tap`, `KS_onion_tap`: A 1024 bit RSA key used for the obsolete `TAP` circuit extension handshake.

Family keys

When a group of relays are controlled by the same operator(s), we call them a "family". A family has a keypair:

- `KP_familyid_ed`, `KS_familyid_ed`: An ed25519 key used to prove membership in a family by signing a family certificate.

Channel authentication

There are other keys that relays use to authenticate as part of their channel negotiation handshakes.

These keys are authenticated with other, longer lived keys. Relays MAY rotate them as often as they like, and SHOULD rotate them frequently—typically, at least once a day.

- `KP_link_ed`, `KS_link_ed`. A short-term Ed25519 "link authentication" key, used to authenticate the link handshake: see "Negotiating and initializing channels". This key is signed by the "signing" key, and should be regenerated frequently.

Legacy channel authentication

These key types were used in older versions of the channel negotiation handshakes.

- `KP_legacy_linkauth_rsa`, `KS_legacy_linkauth_rsa`: A 1024-bit RSA key, used to authenticate the link handshake. (No longer used in modern Tor.) It played a role similar to `KP_link_ed`.

As a convenience, to describe legacy versions of the link handshake, we give a name to the public key used for the TLS handshake itself:

- `KP_legacy_conn_tls`, `KS_legacy_conn_tls`: A short term key used to for TLS connections. (No longer used in modern Tor.) This was another name for the

server's TLS key, which at the time was required to be an RSA key. It was used in some legacy handshake versions.

Creating circuits

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

1. Choose an onion router as an end node (R_N):
    - N MAY be 1 for non-anonymous directory mirror, introduction point, or service rendezvous connections.
    - N SHOULD be 3 or more for anonymous connections. Some end nodes accept streams (see "Opening streams"), others are introduction or rendezvous points (see the Rendezvous Spec).
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/CREATE2 cell along the connection, to be received by the first onion router.
5. Wait until a CREATED/CREATED2 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 client 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/EXTEND2 message along the circuit (see "EXTEND and EXTENDED messages" and "Routing relay cells").
3. When a relay EXTENDED/EXTENDED2 message is received, verify the handshake, and calculate the shared keys. The circuit is now extended.

When an onion router receives an EXTEND relay message, it sends a CREATE cell to the next onion router, with the enclosed onion skin as its body.

encrypt the stream of data going from the client to the relay, and Kb is used to encrypt the stream of data going from the relay to the client.

KDF-RFC5869

For newer KDF needs, including `ntor` and `hs-ntor`. 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` (a constant), and `IKM == secret_input` (the output of the ntor handshake). m_expand and t_key are constant parameters, whose values are stated whenever the use of KDF-RFC5869 is specified.

When partitioning this keystream for the current relay cell encryption protocol from the ntor handshake, the first `SHA1_LEN` bytes form the forward digest Df; the next `SHA1_LEN` form the backward digest Db; the next `KEY_LEN` form Kf, the next `KEY_LEN` form Kb, and the final `SHA1_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.

Channels

A channel is a direct encrypted connection between two Tor relays, or between a client and a relay.

Channels are implemented as TLS sessions over TCP.

Clients and relays may both open new channels; only a relay may be the recipient of a channel.

> Historical note: in some older documentation, channels were sometimes called "connections". This proved to be confusing, and we are trying not to use the term.

As part of establishing a channel, the responding relay will always prove cryptographic ownership of one or more **relay identities**, using a handshake that combines TLS facilities and a series of Tor cells.

As part of this handshake, the initiator MAY also prove cryptographic ownership of its own relay identities, if it has any: public relays SHOULD prove their identities when they initiate a channel, whereas clients and bridges SHOULD NOT do so.

Parties should usually reuse an existing channel rather than opening new a channel to the same relay. There are exceptions here; we discuss them more below.

To open a channel, a client or relay must know the IP address and port of the target relay. (This is sometimes called the "OR address" or "OR port" for the relay.) In most cases, the participant will also know one or more expected identities for the target relay, and will reject the channel if the target relay cannot cryptographically prove ownership of those identities.

> (When initiating a connection, if a reasonably live consensus is available, then the expected identity key is taken from that consensus. But when initiating a connection otherwise, the expected identity key is the one given in the hard-coded authority or fallback list. Finally, when creating a connection because of an EXTEND/EXTEND2 message, the expected identity key is the one given in the message.)

Opening a channel is multi-step process:

1. The initiator opens a new TLS session with certain properties, and the responding relay checks and enforces those properties.
2. Both parties exchange cells over this TLS session in order to establish their identity or identities.
3. Both parties verify that the identities that they received are the ones that they expected. (If any expected key is missing or not as expected, the party MUST close the connection.)

Once this is done, the channel is Open, and regular cells can be exchanged.

Channel lifetime

Channels are not permanent. Either side MAY close a channel if there are no circuits running on it and an amount of time (KeepalivePeriod, defaults to 5 minutes) has passed since the last time any traffic was transmitted over it. 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.

Setting circuit keys

As a final step in creating or extending a circuit, both parties derive a shared set of circuit keys used to encrypt, decrypt, and authenticate relay cells sent over that circuit.

To do this, the parties first use a key expansion algorithm to derive a long (possibly unlimited) keystream from the output of the generator, and then partition the output of that keystream into the necessary circuit keys.

The exact key extension algorithm used, and the format of the partitioned keys, depends on which circuit extension handshake is in use.

KDF-TOR

This key derivation function is used by the CREATE_FAST handshake, and by the obsolete TAP handshake. 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 a stream of derivative key data as

```
K = SHA1(K0 | \[00\]) | SHA1(K0 | \[01\]) | SHA1(K0 | \[02\]) | ...
```

Note that because of the one-byte counter used in each SHA1 input, this KDF MUST NOT be used to generate more than `SHA1_LEN * 256 = 5120` bytes of output. We never approach this amount in practice.

When partitioning this keystream for the current relay cell encryption protocol, the first `SHA1_LEN` bytes of K form KH; the next `SHA1_LEN` form the forward digest Df; the next `SHA1_LEN` 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 client to the relay, and Db seeds the integrity-checking hash for the data stream from the relay to the client. Kf is used to

(Discarding degenerate keys is critical for security; if bad keys are not discarded, an attacker can substitute the relay's CREATED cell's `g^y` with `0` or `1`, thus creating a known `g^xy` and impersonating the relay. 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.

Before using the circuit, the client must verify that the KH value it has received from the relay matches the KH value that it computed from the KDF.

> Note that this "KH" value is unsuitable for some use cases elsewhere in the protocol. That's fine: This handshake is obsolete, deprecated, and unsupported by modern implementations.

TAP's bad hybrid encryption algorithm

The description of TAP above refer to the "legacy hybrid encryption" of a byte sequence M with a public key KP. It is computed as follows:

```
      1. If the length of M is no more than KP_ENC_LEN-KP_PAD_LEN,
         pad and encrypt M with KP.
      2. Otherwise, generate a KEY_LEN byte random key K.
         Let M1 = the first KP_ENC_LEN-KP_PAD_LEN-KEY_LEN bytes of M,
         and let M2 = the rest of M.
         Pad and encrypt K|M1 with KP.  Encrypt M2 with our stream
cipher,
         using the key K.  Concatenate these encrypted values.
```

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. Do not use it as the basis for new protocols! Also note that as used in TAP protocols, case 1 never occurs.

Negotiating and initializing channels

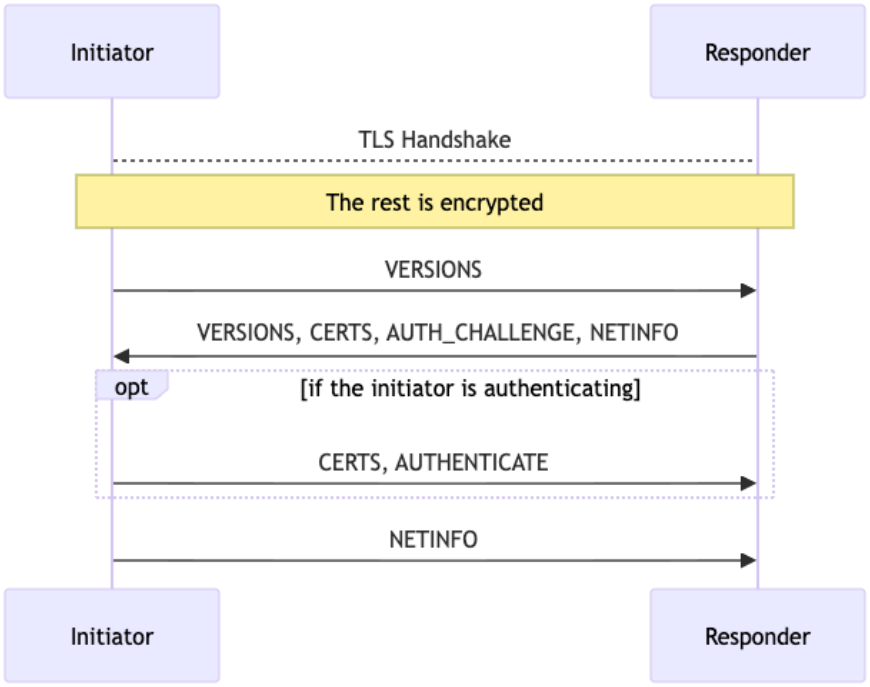
Here we describe the primary TLS behavior used by Tor relays and clients to create a new channel. There are older versions of these handshakes, which we describe in another section.

In brief:

- The initiator starts the handshake by opening a TLS connection.
- Both parties send a VERSIONS to negotiate the link protocol version to use.
- The responder sends a CERTS cell to give the initiator the certificates it needs to learn the responder's identity, an AUTH_CHALLENGE cell that the initiator must include as part of its answer if it chooses to authenticate, and a NETINFO cell to establish clock skew and IP addresses.
- The initiator checks whether the CERTS cell is correct, and decides whether to authenticate.
- If the initiator is not authenticating itself, it sends a NETINFO cell.
- If the initiator is authenticating itself, it sends a CERTS cell, an AUTHENTICATE cell, a NETINFO cell.

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 VPADDING cells.

(The AUTHORIZE cell type is reserved for future use by scanning-resistance designs. It is not specified here.)



The TLS handshake

The initiator must send a ciphersuite list containing at least one ciphersuite other than those listed in the obsolete v1 handshake.

> This is trivially achieved by using any modern TLS implementation, and most implementations will not need to worry about it.
>
> This requirement distinguishes the current protocol (sometimes called the "in-protocol" or "v3" handshake) from the obsolete v1 protocol.

TLS security considerations

(Standard TLS security guarantees apply; this is not a comprehensive guide.)

Implementations SHOULD 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.

nobody at all). It's not very fast and not very good. (See Goldberg's "On the Security of the Tor Authentication Protocol".)

Clients SHOULD NOT use this handshake. Relays SHOULD NOT accept this handshake, and MAY reply to it with a DESTROY cell.

Define `TAP_C_HANDSHAKE_LEN` as `DH_LEN+KEY_LEN+KP_PAD_LEN`. Define `TAP_S_HANDSHAKE_LEN` as `DH_LEN+SHA1_LEN`.

The body 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 encrypted using the "legacy hybrid encryption" algorithm described below, to the server's onion key (`KP_onion_tap`), giving a client handshake:

Field	Size
KP-encrypted:	
- Padding	`KP_PAD_LEN` bytes
- Symmetric key	`KEY_LEN` bytes
- First part of `g^x`	`KP_ENC_LEN-KP_PAD_LEN-KEY_LEN` bytes
Symmetrically encrypted	
- Second part of `g^x`	`DH_LEN-(KP_ENC_LEN-KP_PAD_LEN-KEY_LEN)` bytes

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

Field	Size
DH data (`g^y`)	`DH_LEN` bytes
Derivative key data (`KH`)	`SHA1_LEN` bytes (see "Setting Circuit Keys")

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

Once the handshake between the client and a relay 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.

Clients SHOULD NOT send the second format of CREATE cells (the one with the handshake type tag) to a server directly. Instead, it was only generated to tunnel ntor requests inside a legacy EXTEND cell.

The format of a CREATED cell is:

Field	Description	Size
`HDATA`	Server Handshake Data	`TAP_S_HANDSHAKE_LEN` bytes

(It's equivalent to a CREATED2 cell with length of `TAP_S_HANDSHAKE_LEN`.)

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

EXTEND and EXTENDED messages

The body for an (obsolete) EXTEND relay message consists of:

Field	Size
Address	4 bytes
Port	2 bytes
Onion skin	`TAP_C_HANDSHAKE_LEN` bytes
Identity fingerprint	`SHA1_LEN` bytes

Clients SHOULD NOT send EXTEND messages; relays SHOULD NOT accept them.

The body of an (obsolete) EXTENDED message is the same as the body of a CREATED cell.

Clients SHOULD use the EXTEND format whenever sending a TAP handshake, and MUST use it whenever the EXTEND message 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 message, clients SHOULD use the format with 'client handshake type tag'.

The "TAP" handshake

This obsolete 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

Implementations SHOULD NOT allow TLS compression – although we don't know a way to apply a CRIME-style attack to current Tor directly, it's a waste of resources.

To ensure compatibility:

- All implementations SHOULD support TLS 1.3.
- Relay implementations SHOULD support TLS 1.2 and TLS 1.3.
- With TLS 1.2, all implementations SHOULD support as many of the following ciphersuites and groups as possible:
  - `ECDHE_RSA_WITH_AES_128_GCM_SHA256`
  - `ECDHE_RSA_WITH_CHACHA20_POLY1305`
  - NIST group P-256
  - Curve25519
- With TLS 1.3, all implementations SHOULD support as many of the following ciphers and handshakes as possible:
  - `AES_128_GCM_SHA256`
  - `CHACHA20_POLY1305_SHA256`
  - NIST group P-256
  - X25519

In general, relay implementations SHOULD support as many secure TLS ciphersuites, ciphers, and handshakes as possible.

Relay implementations SHOULD NOT support insecure or deprecated ciphersuites and options.

Negotiating versions with VERSIONS cells

There are multiple instances of the Tor channel protocol.

Once the TLS handshake is complete, both parties send a variable-length VERSIONS cell to negotiate which one they will use.

The body 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 the body contains an odd number of bytes).

Any VERSIONS cells sent after the first VERSIONS cell MUST be ignored. (To be interpreted correctly, later VERSIONS cells MUST have a CIRCID_LEN matching the version negotiated with the first VERSIONS cell.)

> (The obsolete v1 channel protocol does note VERSIONS cells. Implementations MUST NOT list version 1 in their VERSIONS cells. The obsolete v2 channel protocol can only be used after renegotiation; implementations MUST NOT list version 2 in their VERSIONS cells unless they have renegotiated the TLS session.)

The currently specified Link protocols are:

Version	Description
1	(Obsolete) The "certs up front" handshake.
2	(Obsolete) Uses the renegotiation-based handshake. Introduces variable-length cells.
3	(Obsolete) Begins use of the current ("in-protocol") handshake.
4	Increases circuit ID width to 4 bytes.
5	Adds support for link padding and negotiation.

CERTS cells

The CERTS cell describes the keys that a Tor instance is claiming to have, and provides certificates to authenticate that those keys belong, ultimately, to one or more identity keys.

CERTS is a variable-length cell. Its body format is:

Field	Size	Description
N	1	Number of certificates in cell
N times:		
- CertType	1	Type of certificate
- CertLen	2	Length of "Certificate" field
- Certificate	CertLen	Encoded certificate

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

The CertType field determines the format of the certificate, and the roles of its keys within the Tor protocol. Recognized values are defined in "Certificate types (CERT_TYPE field)".

Obsolete circuit extension handshakes

Older versions of Tor used a set of non-extensible cells and messages to create and extend circuits.

They also had older cryptographic handshakes to do so.

In this section, we describe those cells, messages, and handshakes.

Clients and relays SHOULD NOT generate or support these.

> Support for these formats was removed entirely in Tor 0.4.9.1-alpha. Once all earlier versions of Tor are unsupported, we can move this section to the "historical" heading.

CREATE and CREATED cells

The CREATE and CREATED cells were older versions of CREATE2 and CREATED2, which could only be used for the obsolete TAP circuit extension protocol.

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

Field	Description	Size
`HDATA`	Client Handshake Data	`TAP_C_HANDSHAKE_LEN` bytes

or

Field	Description	Size
`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. Recognized HTAG values are:

Value	Description
'ntorNTORntorNTOR'	ntor

1. Note that the usage of "2" above is a historical accident. In the future, we should always use the same `EXT_FIELD_TYPE` number for a client's response and the corresponding server reply (if any). ↩ ↩2

A CERTS cell MUST have no more than one certificate of any CertType.

Authenticating the responder from its CERTS

The responder's CERTS cell is as follows:

- The CERTS cell contains exactly one CertType 4 Ed25519 `IDENTITY_V_SIGNING_CERT`.
  - This cert must be self-signed; the signing key must be included in a "signed-with-ed25519-key" extension extension. This signing key is `KP_relayid_ed`. The subject key is `KP_relaysign_ed`.
- The CERTS cell contains exactly one CertType 5 Ed25519 `SIGNING_V_TLS_CERT` certificate.
  - This cert must be signed with `KP_relaysign_ed`. Its subject must be the SHA-256 digest of the TLS certificate that was presented during the TLS handshake.
- All of the certs above must be correctly signed, and not expired.

The initiator must check all of the above. If this is successful the initiator knows that the responder has the identity `KP_relayid_ed`.

> The responder's CERTS cell is meant to prove that the responder posseses one or more relay identities. It does this by containing certificate chains from each relay identity key to the TLS certificate presented during the TLS handshake.

> The responder's ownership of that TLS certificate was already proven during the TLS hadnshake itself.

Validating an initiator's CERTS

When required by other parts of this specification; to prove its identity, the initiator must provide a CERTS cell.

> Recall that not all initiators authenticate themselves; bridges and clients do not prove their identity.

The initiator's CERTS cell must conform to the rules for the responder's CERTS cell (see above, exchanging "initiator" and "responder") except that:

Instead of containg a `SIGNING_V_TLS_CERT`,

- The CERTS cell contains exactly one CertType 6 `SIGNING_V_LINK_AUTH` certificate.
  - This certificate must be signed with `KP_relayid_ed`. (Its subject key is deemed to be `KP_link_ed`.)
- All of the certs above must be correctly signed, and not expired.

The responder must check all of the CERTS cell's properties (as stated here, and in the previous section). If this is successful **and** the initiator later sends a valid AUTHENTICATE cell, then the initiator has ownership of the presented `KP_relayid_ed`.

> Note that the CERTS cell is *not* yet sufficent to authenticate the channel, until AUTHENTICATE is received: unlike the responder, the initiator is not required to present a TLS certificate during the TLS handshake. Therefore, the initiator has no meaningful `SIGNING_V_TLS_CERT` certificate.
>
> Therefore, instead, the initiator's CERTS cell proves a chain from the initiator's relay identities to a "link authentication" key. This key is later used to sign an "authentication challenge", and bind it to the channel.

Authenticating an RSA identity (#auth-RSA)

After processing a CERTS cell to find the other party's `KP_relayid_ed` Ed25519 identity key, a Tor instance MAY *additionally* check the CERTS cell to find the other party's `KP_relayid_rsa` legacy RSA identity key.

A party with a given `KP_relayid_ed` identity key also has a given `KP_relayid_rsa` legacy identity key when all of the following are true. (A party MUST NOT conclude that an RSA identity key is associated with a channel without checking these properties.)

- The CERTS cell contains exactly one CertType 2 `RSA_ID_X509` certificate.
  - This must be a self-signed certificate containing a 1024-bit RSA key; that key's exponent must be 65537. That key is `KP_relayid_rsa`.
- The CERTS cell contains exactly one CertType 7 `RSA_ID_V_IDENTITY` certificate.
  - This certificate must be signed with `KP_relayid_rsa`.
  - This certificate's subject key must be the same as an already-authenticated `KP_relayid_ed`.
- All of the certs above must be correctly signed, not expired, and not before their `validAfter` dates.

Within this extension, capabilities SHOULD be sorted in ascending order by `protocol_id`, then by `protocol_cap_number`. (For example, [01 01] comes before [01 02], which comes before [02 01].)

Not every subprotocol capability is supported with this extension: only a limited list is supported. That list of supported capabilities is:

Name	Value	Encoding
`RELAY_CRYPTO_CGO`	"Relay=6"	[02 06]

A client MUST NOT list any capability in this extension unless all of the following apply:

- The capability is listed in the table above.
- The target supports RELAY_NEGOTIATE_SUBPROTO ("Relay=5").
- The target supports the capability in question.

To determine whether a target relay supports a given capability, the client looks at the relay's supported protocols. If the target relay is not listed in the consensus, the client SHOULD use the required-relay-protocols list from the latest consensus.

To determine whether an onion service supports a given capability, the client should look in the "proto" item in its descriptor.

> In the future, we plan to add other new subprotocol capabilities to the list above.
>
> It is appropriate to do so for capabilities where all of the following properties hold:
>
> - The client needs to select whether the capability is enabled or not at circuit creation time.
> - The server doesn't need the ability to refuse to support the capability while still letting the circuit open.
> - The client and server don't need to negotitate any parameters related to the capability (this would require a separate extension).

> There is never an implicit automatic relationship between capabilities listed in this extension: If capability X requires capability Y, listing capability X does not "count as" listing capability Y unless documented otherwise.

- 2 – CC_FIELD_RESPONSE [Server to client] 1

  Indicates that the relay will use the congestion control of proposal 324, as requested by the client. Not used with INTRODUCE. One byte in length:

  ```
  sendme_inc        [1 byte]
  ```

  (Note that the use of "2" here is a historical accident; in the future, we should always use same number for a request and its response.)

- 2 – POW [Client to server] [1]

  INTRODUCE only; used to provide proof of work for an onion service.

  (Note that the overloading of 2 here is considered a historical accident.)

- 3 – Subprotocol Request [Client to Server]

  Tells the endpoint what subprotocol capabilities to use on the circuit.

Extension handshake: Subprotocol request

This handshake extension is supported by any relay or onion service advertising support for the subprotocol capability RELAY_NEGOTIATE_SUBPROTO ("Relay=5").

A client includes this extension to indicate one or more subprotocol capabilities which the relay or onion service must support and enable. If the responder does not support all of the listed capabilities, or if it does not enable them all, it MUST close the circuit.

The format of this extension is:

Field	Description	Len
Any number of times:		
- protocol_id	Identifier for the protocol	1 byte
- cap_number	Specific capability	1 byte

Values for protocol_id are given in a table under subprotocol versioning. Specific capabilities are identified with a combination of a protocol and a capability number; for example, the Relay=6 capability is represented as the two-byte sequence [02 06].

If the above tests all pass, then any relay which can prove it has the the identity KP_relayid_ed also has the legacy identity KP_relayid_rsa.

AUTH_CHALLENGE cells

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

Field	Size
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.

The Challenge field is a randomly generated binary string that the initiator must sign (a hash of) as part of their AUTHENTICATE cell.

The Methods are a list of authentication methods that the responder will accept. These methods are defined:

Type	Method
[00 01]	RSA-SHA256-TLSSecret (Obsolete)
[00 02]	(Historical, never implemented)
[00 03]	Ed25519-SHA256-RFC5705

AUTHENTICATE cells

To authenticate, an initiator MUST it respond to the AUTH_CHALLENGE cell with a CERTS cell and an AUTHENTICATE cell.

> Recall that initiators are not always required to authenticate.
>
> (As discussed above, the initiator's CERTS cell differs slightly from what a responder would send.)

An AUTHENTICATE cell contains the following:

Field	Size
AuthType	2
AuthLen	2
Authentication	AuthLen

Responders MUST ignore extra bytes at the end of an AUTHENTICATE cell.

The `AuthType` value corresponds to one of the authentication methods. The initiator MUST NOT send an AUTHENTICATE cell whose AuthType was not contained in the responder's AUTH_CHALLENGE.

An initiator MUST NOT send an AUTHENTICATE cell before it has verified the certificates presented in the responder's CERTS cell, and authenticated the responder.

Link authentication type 3: Ed25519-SHA256-RFC5705

If AuthType is `[00 03]`, meaning "Ed25519-SHA256-RFC5705", the Authentication field of the AUTHENTICATE cell is as follows

Modified values and new fields below are marked with asterisks.

Field	Size	Summary
`TYPE`	8	The nonterminated string `AUTH0003`
`CID`	32	`SHA_256(DER(KP_relayid_rsa))` for initiator
`SID`	32	`SHA_256(DER(KP_relayid_rsa))` for responder
`CID_ED`	32	`KP_relayid_ed` for initiator
`SID_ED`	32	`KP_relayid_ed` for responder
`SLOG`	32	Responder log digest, SHA-256
`CLOG`	32	Initiator log digest, SHA-256
`SCERT`	32	SHA256 of responder's TLS certificate
`TLSSECRETS`	32	RFC5705 information
`RAND`	24	Random bytes
`SIG`	64	Ed25519 signature

- The `TYPE` string distinguishes this authentication document from others. It must be the nonterminated 8-byte string `AUTH0003`.
- For `CID` and `SID`, the SHA-256 digest of an RSA key is computed over the DER encoding of the key.

Field	Size
- `EXT_FIELD_TYPE`	one byte
- `EXT_FIELD_LEN`	one byte
- `EXT_FIELD`	`EXT_FIELD_LEN` bytes

(`EXT_FIELD_LEN` may be zero, in which case `EXT_FIELD` is absent.)

All parties MUST reject messages that are not well-formed per the rules above.

Parties MUST ignore extensions with `EXT_FIELD_TYPE` bodies they do not recognize.

Unless otherwise specified in the documentation for an extension type:

- Each extension type SHOULD be sent only once in a message.
- Parties MUST ignore any occurrence of an extension with a given type after the first such occurrence.
- Extensions SHOULD be sent in numerically ascending order by type.

> (The above extension sorting and multiplicity rules are only defaults; they may be overridden in the description of individual extensions.)

An extension may be supported in CREATE2/CREATED2 messages, in INTRODUCE messages, or both.

> Note that, since the `hs_ntor` handshake does not currently support encrypted data in its relay, there are no Server->Client messages used with `hs_ntor`.

Currently supported extensions are as follows:

Type	Sent by	Name	Create?	Introduce?
1	Client	`CC_FIELD_REQUEST`	Y	Y
2	Service	`CC_FIELD_RESPONSE`	Y	N
2	Client	`POW`	N	Y
3	Client	`SUBPROTO`	Y	Y

- 1 – `CC_FIELD_REQUEST` [Client to server]

  Contains an empty body. Signifies that the client wants to use the extended congestion control described in proposal 324.

CREATE_FAST/CREATED_FAST cells

When creating a one-hop circuit, the client has already established the relay's identity and negotiated a secret key using TLS. Because of this, it is not necessary for the client to perform the public key operations to create a circuit. In this case, the client MAY send a CREATE_FAST cell instead of a CREATE/CREATE2 cell. The relay responds with a CREATED_FAST cell, and the circuit is created.

> In particular, CREATE_FAST is useful when establishing a one-hop circuit in order to download directory information, when the client may not know any onion keys (e.g `KP_onion_ntor`) for the directory.

A CREATE_FAST cell contains:

Field	Size
Key material (`X`)	`SHA1_LEN` bytes

A CREATED_FAST cell contains:

Field	Size
Key material (`Y`)	`SHA1_LEN` bytes
Derivative key data	`SHA1_LEN` bytes (See KDF-TOR)

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.

Parties SHOULD NOT use CREATE_FAST except for creating one-hop circuits.

Sending extensions in the circuit extension handshake

Some handshakes (currently ntor-v3 defined above, and hs-ntor as used for onion services) allow the client or the relay to send additional data as part of the handshake. This additional data must have the following format:

Field	Size
`N_EXTENSIONS`	one byte
`N_EXTENSIONS` times:	

- The `SLOG` field is computed as the SHA-256 digest of all bytes received within the TLS channel up to and including the AUTH_CHALLENGE cell.
  - This includes the VERSIONS cell, the CERTS cell, the AUTH_CHALLENGE cell, and any padding cells.
- The `CLOG` field is computed as the SHA-256 digest of all bytes sent within the TLS channel up to but not including the AUTHENTICATE cell.
  - This includes the VERSIONS cell, the CERTS cell, and any padding cells.
- The `SCERT` field holds the SHA-256 digest of the X.509 certificate presented by the responder as part of the TLS negotiation.
- The `TLSSECRETS` field is computed as the output of a Keying Material Exporter function on the TLS section.
  - The parameters for this exporter are:
    - Label string: "EXPORTER FOR TOR TLS CLIENT BINDING AUTH0003"
    - Context value: The `CID` value above.[1]
    - Length: 32.
  - For keying material exporters on TLS 1.3, see RFC 8446 Section 7.5.
  - For keying material exporters on older TLS versions, see RFC5705.
- The `RAND` field is a uniform squence of Random bytes.
- The `SIG` field is an Ed25519 signature of all earlier members in the Authentication (from `TYPE` through `RAND`) using `KS_link_ed`.

To check an 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 responder MUST ignore any extra bytes in the signed data after the RAND field.

NETINFO cells

To finish the handshake, each party sends the other a NETINFO cell.

A NETINFO cell's body is:

Field	Description	Size
TIME	Timestamp	4 bytes
OTHERADDR:	Other party's address	
- ATYPE	Address type	1 byte
- ALEN	Address length	1 byte
- AVAL	Address value	ALEN bytes

Field	Description	Size
NMYADDR	Number of this party's addresses	1 byte
NMYADDR times:		
- ATYPE	Address type	1 byte
- ALEN	Address length	1 byte
- AVAL	Address value	ALEN bytes

Recognized address types (ATYPE) are:

ATYPE	Description
0x04	IPv4
0x06	IPv6

Implementations SHOULD ignore addresses with unrecognized types.

ALEN MUST be 4 when ATYPE is 0x04 (IPv4) and 16 when ATYPE is 0x06 (IPv6). If the ALEN value is wrong for the given ATYPE value, then the provided address should be ignored.

The `OTHERADDR` field SHOULD be set to the actual IP address observed for the other party.

> (This is typically the address passed to `connect()` when acting as the channel initiator, or the address received from `accept()` when acting as the channel responder.)

In the `ATYPE`/`ALEN`/`AVAL` fields, relays SHOULD send the addresses that they have advertised in their router descriptors. Bridges and clients SHOULD send none of their own addresses.

For the `TIME` field, relays send a (big-endian) integer holding the number of seconds since the Unix epoch. Clients SHOULD send `[00 00 00 00]` as their timestamp, to avoid fingerprinting.

> See proposal 338 for a proposal to extend the timestamp to 8 bytes.

Implementations MUST ignore unexpected bytes at the end of the NETINFO cell.

Using information from NETINFO cells

Implementations MAY use the timestamp value to help decide if their clocks are skewed.

```
Xy = EXP(X,y)
secret_input = Xy | Xb | ID | B | X | Y | PROTOID | ENCAP(VER)
ntor_key_seed = H_key_seed(secret_input)
verify = H_verify(secret_input)

RAW_KEYSTREAM = KDF_final(ntor_key_seed)
(ENC_KEY, KEYSTREAM) = PARTITION(RAW_KEYSTREAM, ENC_KEY_LKEN, ...)

encrypted_msg = ENC(ENC_KEY, SM)

auth_input = verify | ID | B | Y | X | MAC | ENCAP(encrypted_msg) |
    PROTOID | "Server"
AUTH = H_auth(auth_input)
```

The relay then sends as its Created handshake:

Field	Value	Size
`Y`	`Y`	`PUB_KEY_LEN` bytes
`AUTH`	`AUTH`	`DIGEST_LEN` bytes
`MSG`	`encrypted_msg`	`len(SM)` bytes, up to end of the message

Upon receiving this handshake, the client computes:

```
Yx = EXP(Y, x)
secret_input = Yx | Bx | ID | B | X | Y | PROTOID | ENCAP(VER)
ntor_key_seed = H_key_seed(secret_input)
verify = H_verify(secret_input)

auth_input = verify | ID | B | Y | X | MAC | ENCAP(MSG) |
    PROTOID | "Server"
AUTH_expected = H_auth(auth_input)
```

If `AUTH_expected` is equal to `AUTH`, then the handshake has succeeded. The client can then calculate:

```
RAW_KEYSTREAM = KDF_final(ntor_key_seed)
(ENC_KEY, KEYSTREAM) = PARTITION(RAW_KEYSTREAM, ENC_KEY_LKEN, ...)

SM = DEC(ENC_KEY, MSG)
```

SM is the message from the relay, and the client uses KEYSTREAM to generate the shared secrets for the newly created circuit.

Now both parties share the same KEYSTREAM, and can use it to generate their circuit keys.

The client then sends, as its Create handshake:

Field	Value	Size
NODEID	ID	ID_LEN bytes
KEYID	B	PUB_KEY_LEN bytes
CLIENT_PK	X	PUB_KEY_LEN bytes
MSG	encrypted_msg	len(CM) bytes
MAC	msg_mac	MAC_LEN bytes

The client remembers x, X, B, ID, Bx, and msg_mac.

When the server receives this handshake, it checks whether NODEID is as expected, and looks up the (b,B) keypair corresponding to KEYID. If the keypair is missing or the NODEID is wrong, the handshake fails.

Now the relay uses X=CLIENT_PK to compute:

```
Xb = EXP(X,b)
secret_input_phase1 = Xb | ID | X | B | PROTOID | ENCAP(VER)
phase1_keys = KDF_msgkdf(secret_input_phase1)
(ENC_K1, MAC_K1) = PARTITION(phase1_keys, ENC_KEY_LEN, MAC_KEY_LEN)

expected_mac = MAC_msgmac(MAC_K1, ID | B | X | MSG)
```

If expected_mac is not MAC, the handshake fails. Otherwise the relay computes CM as:

```
CM = DEC(MSG, ENC_K1)
```

The relay then checks whether CM is well-formed, and in response composes SM, the reply that it wants to send as part of the handshake. It then generates a new ephemeral keypair:

```
y,Y = KEYGEN()
```

and computes the rest of the handshake:

Initiators MAY use "other relay's address" field to help learn which address their connections may be originating from, if they do not know it; and to learn whether the peer will treat the current connection as canonical. (See Canonical connections)

Implementations SHOULD NOT trust these values unconditionally, especially when they come from non-authorities, since the other party can lie about the time or the IP addresses it sees.

Initiators SHOULD use "this relay's address" to make sure that they have connected to another relay at its canonical address.

---

1. Note: we originally intended that the context value for the TLS exporter would be the initiator's KP_relayid_ed. We implemented it incorrectly, however. Fortunately, we do not believe that the current behavior (using CID instead) affects the security of the protocol. ↩

Obsolete channel handshakes

These handshake variants are no longer in use. Channel initiators MUST NOT send them. Relays MAY detect and reject them. Relays SHOULD NOT accept them.

More specifically:

- Parties SHOULD NOT make any protocol decisions based on which ciphersuites are presented by the TLS initiator. (Other than rejecting the handshake if no secure ciphersuite is available.)
- Parties SHOULD NOT make any protocol decisions based on whether the initiator presents any certificates, or whether the responder presents any certificates beyond the first.

Conformance note:

As of 19 March 2025, existing C Tor implementation still detects and accepts some of these variants. We will remove them.

If you are experienced with TLS, you will find some aspects of this handshake strange or obfuscated. Several historical factors led to its current state.

First, before the development of pluggable transports, Tor tried to avoid censorship by mimicking the behavior of a web client negotiating HTTPS with a web server. If we wanted a secure option that was not in common use, we had to hide our use of that option.

Second, prior to the introduction of TLS 1.3, many more aspects of the handshake (such as the number and nature of certificates sent by each party) were sent in the clear, and were easy to distinguish.

Third, prior to the introduction of TLS 1.3, there was no good encrypted signalling mechanism that a client could use to declare how it wanted the rest of the TLS handshake to proceed. Thus, we wound up using the client's list of supported ciphersuites to send a signal about which variation of the handshake is in use.

Version 1, or "certificates up front"

With this obsolete handshake, the responding relay proves ownership of an RSA identity (`KP_relayid_rsa`), and the initiator also proves ownership of an RSA identity.

```
PROTOID = "ntor3-curve25519-sha3_256-1"
t_msgkdf = PROTOID | ":kdf_phase1"
t_msgmac = PROTOID | ":msg_mac"
t_key_seed = PROTOID | ":key_seed"
t_verify = PROTOID | ":verify"
t_final = PROTOID | ":kdf_final"
t_auth = PROTOID | ":auth_final"

`ENCAP(s)` -- an encapsulation function.  We define this
as `htonll(len(s)) | s`.  (Note that `len(ENCAP(s)) = len(s) + 8`).

`PARTITION(s, n1, n2, n3, ...)` -- a function that partitions a
bytestring `s` into chunks of length `n1`, `n2`, `n3`, and so
on. Extra data is put into a final chunk.  If `s` is not long
enough, the function fails.

H(s, t) = SHA3_256(ENCAP(t) | s)
MAC(k, msg, t) = SHA3_256(ENCAP(t) | ENCAP(k) | s)
KDF(s, t) = SHAKE_256(ENCAP(t) | s)
ENC(k, m) = AES_256_CTR(k, m)

EXP(pk,sk), KEYGEN: defined as in curve25519

DIGEST_LEN = MAC_LEN = MAC_KEY_LEN = ENC_KEY_LEN = PUB_KEY_LEN = 32

ID_LEN = 32  (representing an ed25519 identity key)

For any tag "t_foo":
   H_foo(s) = H(s, t_foo)
   MAC_foo(k, msg) = MAC(k, msg, t_foo)
   KDF_foo(s) = KDF(s, t_foo)
```

Other notation is as in the ntor description above.

The client begins by knowing:

```
B, ID -- The curve25519 onion key (KP_onion_tap)
         and Ed25519 ID (KP_relayid_ed)
         of the server that it wants to use.
CM -- A message it wants to send as part of its handshake.
VER -- An optional shared verification string:
```

The client computes:

```
x,X = KEYGEN()
Bx = EXP(B,x)
secret_input_phase1 = Bx | ID | X | B | PROTOID | ENCAP(VER)
phase1_keys = KDF_msgkdf(secret_input_phase1)
(ENC_K1, MAC_K1) = PARTITION(phase1_keys, ENC_KEY_LEN, MAC_KEY_LEN)
encrypted_msg = ENC(ENC_K1, CM)
msg_mac = MAC_msgmac(MAC_K1, ID | B | X | encrypted_msg)
```

Field	Value	Size
SERVER_KP	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 "KDF-RFC5869" and the tag `m_expand`.

The "ntor-v3" handshake

This handshake extends the ntor handshake to include support for extra data transmitted as part of the handshake. Both the client and the server can transmit extra data; in both cases, the extra data is encrypted, but only server data receives forward secrecy.

To advertise support for this handshake, servers advertise the "Relay=4" subprotocol. To select it, clients use the 'ntor-v3' HTYPE value in their CREATE2 cells.

In this handshake, we define:

(If the initiator does not have an RSA identity to prove, it invents one and throws it away afterwards.)

To select this handshake, the initiator starts a TLS handshake containing no ciphersuites other than these:

```
TLS_DHE_RSA_WITH_AES_256_CBC_SHA
TLS_DHE_RSA_WITH_AES_128_CBC_SHA
SSL_DHE_RSA_WITH_3DES_EDE_CBC_SHA
```

> Note that because of this list, it is impossible to use this obsolete handshake with TLS 1.3.

As part of the TLS handshake, the initiator sends a two-certificate chain, consisting of an X.509 certificate for its short-term connection public key (`KP_legacy_conn_tls`) signed by `KP_relayid_rsa`, and a second self-signed X.509 certificate containing `KP_relayid_rsa`. The responder sends a similar certificate chain.

Once the TLS handshake is done, both parties validate each other's certificate chains. If they are valid, then the connection is Open, and both parties may start exchanging cells.

Version 2, or "renegotiation"

In "renegotiation" (a.k.a. "the v2 handshake"), the connection initiator selects at least one ciphersuite not in the list above. The initiator sends no certificates, and the responder sends a single connection certificate in return.

(If the responder sends a certificate chain, the initiator assumes that it only knows about the v1 handshake.)

Once this initial TLS handshake is complete, the initiator renegotiates the session. During the renegotiation, each party sends a two-certificate chain as in the "certificates up front" handshake above.

When this handshake is used, both parties immediately send a VERSIONS cell, and after negotiating a link protocol version (which will be 2), each sends a NETINFO cell to confirm their addresses and timestamps. At that point, the channel is Open. No other intervening cell types are allowed.

Indicating support for the in-protocol handshake

When the in-protocol handshake was new, we placed a set of constraints on the certificate that the responder would send to indicate that it supported the v3 handshake.

Specifically, if at least one of these properties was true of the responders's certificate, the initiator could be sure that the responder supported the in-protocol handshake:

- 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.

Otherwise, the initiator would assume that only the v2 protocol was in use.

Fixed ciphersuite list

For a long time, clients would advertise a certain "fixed ciphersuite list" regardless of whether they actually supported those ciphers.

That list is:

> In practice, modern Tor clients *always* have extensions to send, and all relays provide ntor-v3, so clients will always use ntor-v3.

[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"
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
EXP(a, b) = The ECDH algorithm for establishing a shared secret.
```

To perform the handshake, the client needs to know `NODEID = SHA1(DER(KP_relayid_id))` for the server, and an ntor onion key (a curve25519 public key, `KP_onion_ntor`) 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:

Field	Value	Size
`NODEID`	Server identity digest	`ID_LENGTH` bytes
`KEYID`	KEYID(B)	`H_LENGTH` bytes
`CLIENT_KP`	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:

that it is indeed connected to the correct target relay, and prevents certain man-in-the-middle attacks.

The "legacy identity" and "identity fingerprint" fields are computed as `SHA1(DER(KP_relayid_rsa))`.

The extending relay MUST check *all* provided identity keys (if they recognize the format), and and MUST NOT extend the circuit if the target relay did not prove its ownership of any such identity key. If only one identity key is provided, but the extending relay knows the other (from directory information), then the relay SHOULD also enforce the key in the directory.

If the extending relay has a channel with a given Ed25519 ID and RSA identity, and receives a request for that Ed25519 ID and a different RSA identity, it SHOULD NOT attempt to make another connection: it should just fail and DESTROY the circuit.

The client MAY include multiple IPv4 or IPv6 link specifiers in an EXTEND2 message; current relay implementations only consider the first of each type.

After checking relay identities, extending relays generate a CREATE2 cell from the contents of the EXTEND2 message. See Creating circuits for details.

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

[Support for EXTEND2/EXTENDED2 was added in Tor 0.2.4.8-alpha.]

When generating an EXTEND2 message, clients SHOULD include the target's Ed25519 identity whenever the target has one, and whenever the target supports the subprotocol "LinkAuth=3" (`LINKAUTH_ED25519_SHA256_EXPORTER`). (See LinkAuth).

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.

Clients should only use the ntor handshake when they have no extensions to send. When a client *does* have extensions, it MUST use ntor-v3.

```
      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.
```

When encountering this list, a responder would not select any ciphersuites besides the mandatory-to-implement 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.

Clients no longer report ciphers that they do not support.

Legacy CERTS authentication: Responder has RSA Identity only

A Tor relay that has only an RSA identity key (`KP_relayid_rsa`) and not an Ed25519 identity key (`KP_relayid_ed`) will present a different set of certificates in its CERTS cell.

(Relays like this are no longer supported; all relays must now have Ed25519 identities.)

To authenticate a responder as having only an RSA identity, the initiator would verify the following:

- The CERTS cell contains exactly one CertType 2 `RSA_ID_X509` certificate.
  - This must be a self-signed certificate containing a 1024-bit RSA key; that key's exponent must be 65537. That key is `KP_relayid_rsa`.
- The CERTS cell contains exactly one CertType 1 `TLS_LINK_X509` certificate.
  - It must be signed with `KP_relayid_rsa`.
  - Its subject key must be the same as `KP_legacy_conn_tls` (the key used to negotiate the TLS connection).
- All of the certs above must be correctly signed, not expired, and not before their `validAfter` dates.

Legacy CERTS authentication: Initiator has RSA Identity only

As discussed in "Validating an initiator's CERTS", the initiator of the v3 handshake does not present a TLS certificate.

Therefore, to process an initiator's CERTS cell, the responder would have to procede as for a responder's certificates, as described above, except that **instead** of checking for a `TLS_LINK_X509` certificate, it would need to verify that:

- The CERTS cell contains exactly one CertType 3 `LINK_AUTH_X509` certificate.
  - This certificate must be signed with `KP_relayid_rsa`. Its subject key is deemed to be `KP_legacy_linkauth_rsa`.
- All of the certs above must be correctly signed, not expired, and not before their `validAfter` dates.

Link authentication type 1: RSA-SHA256-TLSSecret

This is an obsolete authentication method used before RFC5705 support was ubiquitous. It is nearly the same as Ed25519-SHA256-RFC5705, but lacks support for Ed25519, and does not use keying material exporters (which were not widely supported at the time it as used.

If AuthType is `[00 01]` (meaning "RSA-SHA256-TLSSecret"), then the authentication field of the AUTHENTICATE cell contains the following:

Field	Size	Description
`TYPE`	8	The nonterminated string `AUTH0001`
`CID`	32	`SHA256(KP_relayid_rsa)` for initiator

Extend and Extended messagess

To extend an existing circuit, the client sends an EXTEND2 message, in a RELAY_EARLY cell, to the last node in the circuit.

The body of an EXTEND2 message contains:

Field	Description	Size
`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:

Value	Description
[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 SHA-1 identity fingerprint. At most one may be listed.
[03]	Ed25519 identity. A 32-byte Ed25519 identity. At most one may be listed.

Nodes MUST ignore unrecognized specifiers, and MUST accept multiple instances of specifiers other than 'legacy identity' and 'Ed25519 identity'. (Nodes SHOULD reject link specifier lists that include multiple instances of either one of those specifiers.)

For purposes of indistinguishability, implementations SHOULD send these link specifiers, if using them, in this order: [00], [02], [03], [01].

The "Ed25519 identity" field is the Ed25519 identity key (`KP_relayid_ed`) of the target node. Including this key information allows the relay extending to verify

Value	Description
	ntor-v3 – ntor extended with extra data; see The “ntor-v3” handshake

Relays always respond to a successful CREATE2 with a CREATED2. On failure, the relay sends a DESTROY to shut down the circuit.

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

Choosing circuit IDs in create cells

The CircID for a CREATE2 cell is a nonzero integer, selected by the node (client or relay) that sends the CREATE2 cell. Depending on the link protocol version, there are certain rules for choosing the value of CircID which MUST be obeyed, as implementations MAY decide to refuse in case of a violation. In link protocol 3 or lower, CircIDs are 2 bytes long; in protocol 4 or higher, CircIDs are 4 bytes long.

In link protocol version 3 or lower, the nodes choose from only one half of the possible values based on the relays’ public identity keys, in order to avoid collisions. 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.) A client with no public key MAY choose any CircID it wishes, since clients never need to process CREATE2 cells.

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

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.

Existing Tor implementations choose their CircID values at random from among the available unused values. To avoid distinguishability, new implementations should do the same. Implementations MAY give up and stop attempting to build new circuits on a channel, if a certain number of randomly chosen CircID values are all in use (today’s Tor stops after 64).

Field	Size	Description
`SID`	32	`SHA256(KP_relayid_rsa)` for responder
`SLOG`	32	SHA256 of responder transcript
`CLOG`	32	SHA256 of initiator transcript
`SCERT`	32	SHA256 of responder’s TLS certificate
`TLSSECRETS`	32	An ad-hoc HMAC output
`RAND`	24	Random bytes
`SIG`	Variable	RSA signature

Notes are as for Ed25519-SHA256-RFC5705, except as follows:

- The `TLSSECRETS` fields holds a SHA256 HMAC, using the TLS master secret as the secret key, of the following concatenated fields:
  - `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"`
- The `SIG` fields holds an RSA signature of a SHA256 hash of all the previous fields (that is, `TYPE` through `RAND`), using the initiator’s `KS_legacy_linkauth_rsa`. This field extends through the end of the AUTHENTICATE cell.

Responders MUST NOT accept this AuthType if the initiator has claimed to have an Ed25519 identity.

Cells (messages on channels)

The basic unit of communication on a Tor channel is a "cell".

Once a TLS connection is established, the two parties send cells to each other. Cells are sent serially, one after another.

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.

Most cells are of fixed length, with the actual length depending on the negotiated link protocol on the channel. Below we designate the negotiated protocol as `v`.

As an exception, `VERSIONS` cells are always sent with `v = 0`, since no version has yet been negotiated.

A fixed-length cell has this format:

Field	Size in bytes	Notes
`CircID`	`CIRCID_LEN(v)`	
`Command`	1	
`Body`	`CELL_BODY_LEN`	Padded to fit

The value of `CIRCID_LEN` depends on the negotiated link protocol.

Some cells have variable length; the length of these cells is encoded in their header.

A variable-length cell has this format:

Field	Size in bytes	Notes
`CircID`	`CIRCID_LEN(v)`	
`Command`	1	
`Length`	2	A big-endian integer
`Body`	`Length`	

Creating and extending circuits

Users set up circuits incrementally, one hop at a time. To create a new circuit, clients send a CREATE2 cell to the first node, with the first half of an authenticated handshake; that node responds with a CREATED2 cell with the second half of the handshake. To extend a circuit past the first hop, the client sends an EXTEND2 client relay message (see Extend and Extended messages which instructs the last node in the circuit to send a CREATE2 cell to extend the circuit.

In addition to CREATE2 and CREATED2 cells, there are also:

- An obsolete "CREATE/CREATED format used in old versions of Tor.
- A specialized "CREATE_FAST/CREATED_FAST" format for one hop circuits.

Create and Created cells

A CREATE2 cell contains:

Field	Description	Size
`HTYPE`	Client Handshake Type	2 bytes
`HLEN`	Client Handshake Data Len	2 bytes
`HDATA`	Client Handshake Data	`HLEN` bytes

A CREATED2 cell contains:

Field	Description	Size
`HLEN`	Server Handshake Data Len	2 bytes
`HDATA`	Server Handshake Data	`HLEN` bytes

Recognized HTYPEs (handshake types) are:

Value	Description
0x0000	TAP – the original (obsolete) Tor handshake; see The "TAP" handshake
0x0001	reserved
0x0002	ntor – the ntor+curve25519+sha256 handshake; see The "ntor" handshake
0x0003	

Circuit management

This section describes how circuits are created, and how they operate once they are constructed.

Fixed-length and variable-length cells are distinguished based on the value of their Command field:

- Command 7 (`VERSIONS`) is variable-length.
- Every other command less than 128 denotes a fixed-length cell.
- Every command greater than or equal to 128 denotes a variable-length cell.

> Historical note:
>
> On version 1 connections, all cells were fixed-length.
>
> On version 2 connections, only the `VERSIONS` command was variable-length, and all others were fixed-length.
>
> These link protocols are obsolete, and implementations SHOULD NOT support them.

Interpreting the fields: CircID

The `CircID` field determines which circuit, if any, the cell is associated with. If the cell is not associated with any circuit, its `CircID` is set to 0.

> Note that a CircID is a channel-local identifier.
>
> A single multi-hop circuit will have a different CircID on every channel that is used to transmit its data.

Interpreting the fields: Command

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

Value	C	P	Identifier	Description
0	N		`PADDING`	Link Padding
1	Y		`CREATE`	Create circuit (deprecated)
2	Y		`CREATED`	Acknowledge CREATE (deprecated)
3	Y		`RELAY`	End-to-end data
4	Y		`DESTROY`	Destroy circuit
5	Y		`CREATE_FAST`	Create circuit, no public key
6	Y		`CREATED_FAST`	

Value	C	P	Identifier	Description
				Acknowledge CREATE_FAST
8	N		NETINFO	Time and address info
9	Y		RELAY_EARLY	End-to-end data; limited
10	Y		CREATE2	Create circuit
11	Y		CREATED2	Acknowledge CREATED2
12	Y	5	PADDING_NEGOTIATE	Padding negotiation

The variable-length `Command` values are:

Value	C	P	Identifier	Description
7	N		VERSIONS	Negotiate link protocol
128	N		VPADDING	Variable-length padding
129	N		CERTS	Certificates
130	N		AUTH_CHALLENGE	Challenge value
131	N		AUTHENTICATE	Authenticate initiator
132	N	n/a	AUTHORIZE	(Reserved)

In the tables above, **C**=Y indicates that a command must have a nonzero CircId, and **C**=N indicates that a command must have a zero CircId. Where given, **P** is the first link protocol version to support a command. Commands with no value given for **P** are supported at least in link protocols 3 and above.

No other command values are allowed. Implementations SHOULD NOT send undefined command values. Upon receiving an unrecognized command, an implementation MAY silently drop the cell and MAY terminate the channel with an error.

> Extensibility note:
>
> When we add new cell command types, we define a new link protocol version to indicate support for that command.
>
> Therefore, implementations can now safely assume that other correct implementations will never send them an unrecognized cell command.
>
> Historically, before the link protocol was not versioned, implementations would drop cells with unrecognized commands, under the assumption that the command was sent by a more up-to-date version of Tor.

Interpreting the fields: Cell Body

The interpretation of a cell's Body depends on the cell's command. See the links in the command descriptions above for more information on each command.

Padding fixed-length cell bodies

Often, the amount of information to be sent in a fixed-length cell is less than `CELL_BODY_LEN` bytes. When this happens, the sender MUST fill the unused part of the cell's body with zero-valued bytes.

Recipients MUST ignore padding bytes.

> RELAY and RELAY_EARLY cells' bodies contain encrypted data, and are always full from the point of the view of the channel layer.
>
> The *plaintext* of these cells' contents may be padded; this uses a different mechanism and does not interact with cell body padding.

Variable-length cells never have extra space, so there is no need to pad their bodies. Unless otherwise specified, variable-length cells have no padding.