The Onion Routing

A Detailed Handbook

Amirhossein Khajepour, Farid Zandi, Navid Malek

Contents

Tor Specification

1. Preliminaries

1.1 Ciphers

1.1.1 Stream Cipher

For stream ciphers Tor uses 128-bit AES in counter mode, with an IV of all 0 bytes.
In section 10.2 we provide some notes about AES counter mode. Tor usage for AES counter mode with EVP APIs is located here .
For a public-key cipher, uses RSA with 1024-bit keys and a fixed exponent of 65537 and OAEP-MGF1 padding format, with SHA-1 as its digest function. Further information about RSA explained in section 10.3.

```
1
2  src/lib/crypt_ops/crypto_curve25519.c +190 curve25519 key generation
3
4
5      We also use the Curve25519 group and the Ed25519 signature format in
6      several places.
7
8      For Diffie-Hellman, unless otherwise specified, we use a generator
9      (g) of 2.  For the modulus (p), we use the 1024-bit safe prime from
10     rfc2409 section 6.2 whose hex representation is:
11
12        "FFFFFFFFFFFFFFFFC90FDAA22168C234C4C6628B80DC1CD129024E08"
13        "8A67CC74020BBEA63B139B22514A08798E3404DDEF9519B3CD3A431B"
14        "302B0A6DF25F14374FE1356D6D51C245E485B576625E7EC6F44C42E9"
15        "A637ED6B0BFF5CB6F406B7EDEE386BFB5A899FA5AE9F24117C4B1FE6"
16        "49286651ECE65381FFFFFFFFFFFFFFFF"
17
18     As an optimization, implementations SHOULD choose DH private keys (x) of
19     320 bits.  Implementations that do this MUST never use any DH key more
20     than once.
```

```
21      [May other implementations reuse their DH keys?? –RD]
22      [Probably not. Conceivably, you could get away with changing DH keys once
23      per second, but there are too many oddball attacks for me to be
24      comfortable that this is safe. –NM]
25
26      For a hash function, unless otherwise specified, we use SHA–1.
27
28      KEY_LEN=16.
29      DH_LEN=128; DH_SEC_LEN=40.
30      PK_ENC_LEN=128; PK_PAD_LEN=42.
31      HASH_LEN=20.
32
33      We also use SHA256 and SHA3–256 in some places.
34
35      When we refer to "the hash of a public key", unless otherwise
36      specified, we mean the SHA–1 hash of the DER encoding of an ASN.1 RSA
37      public key (as specified in PKCS.1).
38
39      All "random" values MUST be generated with a cryptographically
40      strong pseudorandom number generator seeded from a strong entropy
41      source, unless otherwise noted.
```

2. System Overview

Tor is a distributed overlay network designed to anonymize low-latency TCP-based applications such as web browsing, secure shell, and instant messaging. Clients choose a path through the network and build a circuit, in which each node in the path knows its predecessor and successor, but no other nodes in the circuit. Traffic flowing down the circuit is sent in fixed-size cells, which are unwrapped by a symmetric key at each node (like the layers of an onion) and relayed downstream. Each onion router maintains a long-term identity key and a short-term onion key. The identity key is used to sign TLS certificates, to sign the OR's router descriptor (a summary of its keys, address, bandwidth, exit policy, and so on), and (by directory servers) to sign directories. The onion key is used to decrypt requests from users to set up a circuit and negotiate ephemeral keys. The TLS protocol also establishes a short- term link key when communicating between ORs. Short-term keys are rotated periodically and independently, to limit the impact of key compromise. [4]

Every Tor relay has multiple public/private keypairs. As referenced in chapter 1 any keypairs has been generated with on of these methods: 1024-bit RSA, Curve25519, Ed25519.

1024-bit RSA :

- A long-term signing-only "Identity key" used to sign documents and certificates, and used to establish relay identity.
- A medium-term TAP "Onion key" used to decrypt onion skins when accepting circuit extend attempts.Old keys MUST be accepted for a while after they are no longer advertised. Because of this, relays MUST retain old keys for a while after they're rotated. (See "onion key lifetime parameters" in dir-spec.txt.)
- A short-term "Connection key" used to negotiate TLS connections. Tor implementations MAY rotate this key as often as they like, and SHOULD rotate this key at least once a day. 3.2

Curve25519 key :

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

parameters" in dir-spec.txt.)

Ed25519 :

- A long-term "master identity" key. This key never changes; it is used only to sign the "signing" key below. may be kept offline.
- A medium-term "signing" key. This key is signed by the master identity key, and must be kept online. A new one should be generated periodically. It signs nearly everything else.
- A short-term "link authentication" key, used to authenticate the link handshake: see section 4 below. This key is signed by the "signing" key, and should be regenerated frequently.

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

3. Connections

3.1 Cipher suit

A cipher suite is a set of algorithms that help secure a network connection that uses Transport Layer Security (TLS) or its now-deprecated predecessor Secure Socket Layer (SSL). The set of algorithms that cipher suites usually contain include: a key exchange algorithm, a bulk encryption algorithm, and a message authentication code (MAC) algorithm.[3]
The key exchange algorithm is used to exchange a key between two devices. This key is used to encrypt and decrypt the messages being sent between two machines. The bulk encryption algorithm is used to encrypt the data being sent. The MAC algorithm provides data integrity checks to ensure that the data sent does not change in transit. In addition, cipher suites can include signatures and an authentication algorithm to help authenticate the server and or client.[3]
After coordinating which cipher suite to use, the server and the client still has the ability to change the coordinated ciphers by using the ChangeCipherSpec protocol in the current handshake or in a new handshake.[3]

3.1.1 TLS 1.0–1.2 handshake

This client starts the process by sending a clientHello message to the server that includes the version of TLS being used and a list of cipher suites in the order of the client's preference. In response, the server sends a serverHello message that includes the chosen cipher suite and the session ID. Next the server sends a digital certificate to verify its identity to the client. **The server may also request a client's digital certification if needed**.[3]

If the client and server are not using pre-shared keys, the client then sends an encrypted message to the server that enables the client and the server to be able to compute which secret key will be used during exchanges.[3]
After successfully verifying the authentication of the server and, if needed, exchanging the secret key, the client sends a finished message to signal that it is done with the handshake process. After receiving this message, the server sends a finished message that confirms that the handshake is complete. Now the client and the server are in agreement on which cipher suite to use to communicate with each other.[3]

```
      Client                                           Server

      ClientHello                  -------->
                                                  ServerHello
                                                 Certificate*
                                           ServerKeyExchange*
                                          CertificateRequest*
                               <--------      ServerHelloDone
      Certificate*
      ClientKeyExchange
      CertificateVerify*
      [ChangeCipherSpec]
      Finished                 -------->
                                            [ChangeCipherSpec]
                               <--------             Finished
      Application Data         <------->     Application Data

* - optional
```

Figure 3.1: TLS Handshake Protocol [9]

Key exchange/agreement	Authentication	Block/stream ciphers	Message authentication
RSA	RSA	RC4	Hash-based MD5
Diffie–Hellman	DSA	Triple DES	SHA hash function
ECDH	ECDSA	AES	
SRP		IDEA	
PSK		DES	
		Camellia	

Figure 3.2: Algorithms supported in TLS 1.0–1.2 cipher suites [3]

3.2 Connections

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

There are three ways to perform TLS handshakes with a Tor server.

1. **certificates-up-front**: Both the initiator and responder send a two-certificate chain as part of their initial handshake. (This is supported in all Tor versions.)
2. **renegotiation**: The responder provides a single certificate, and the initiator immediately performs a TLS renegotiation. (This is supported in Tor 0.2.0.21 and later.)
3. **in-protocol**: the initial TLS negotiation completes, and the parties bootstrap themselves to mutual authentication via use of the Tor protocol without further TLS handshaking. (This is supported in 0.2.3.6-alpha and later.)

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

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

1. TLS_DHE_RSA_WITH_AES_256_CBC_SHA
2. TLS_DHE_RSA_WITH_AES_128_CBC_SHA
3. SSL_DHE_RSA_WITH_3DES_EDE_CBC_SHA

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

In "in-protocol" (a.k.a. "the v3 handshake"), the initiator sends no certificates, and the responder sends a single connection certificate. The choice of ciphersuites must be as in a "renegotiation" handshake. There are additionally a set of constraints on the connection certificate, which the initiator can use to learn that the in-protocol handshake is in use.
These are some usefull fields in the structure of an X.509 v3 certificate according to [5]:

Listing 3.1: X.509 v3 Certificate structure

```
1  Certificate  ::=  SEQUENCE  {
2          tbsCertificate          TBSCertificate,
3          signatureAlgorithm      AlgorithmIdentifier,
4          signatureValue          BIT STRING  }
5
6     TBSCertificate  ::=  SEQUENCE  {
7          version           [0]  EXPLICIT Version DEFAULT v1,
8          serialNumber           CertificateSerialNumber,
9          signature              AlgorithmIdentifier,
10         issuer                 Name,
11         validity               Validity,
12         subject                Name,
13         subjectPublicKeyInfo   SubjectPublicKeyInfo,
14         issuerUniqueID    [1]  IMPLICIT UniqueIdentifier OPTIONAL,
15                                 -- If present, version MUST be v2 or v3
16         subjectUniqueID   [2]  IMPLICIT UniqueIdentifier OPTIONAL,
17                                 -- If present, version MUST be v2 or v3
18         extensions        [3]  EXPLICIT Extensions OPTIONAL
19                                 -- If present, version MUST be v3
20         }
```

issuer

The issuer field identifies the entity that has signed and issued the certificate. The issuer field MUST contain a non-empty distinguished name (DN). The issuer field is defined as the X.501 type Name.

Implementations of this specification MUST be prepared to receive the following standard attribute types in issuer and subject (Section 4.1.2.6 of [5]) names:

1. country
2. organization
3. organizational unit
4. distinguished name qualifier (DN)
5. state or province name
6. common name (e.g., "Susan Housley")
7. serial number

Specifically, at least one of these properties must be true of the certificate:

1. The certificate is self-signed
2. Some component other than "commonName" is set in the subject or issuer DN of the certificate.
3. The commonName of the subject or issuer of the certificate ends with a suffix other than ".net".
4. The certificate's public key modulus is longer than 1024 bits.

Here is implementation of in-protocol handshake through the Tor source code:

Listing 3.2: in-protocol implementation line number 287

```
1 int
2 tor_tls_context_init_certificates(tor_tls_context_t *result,
3                                   crypto_pk_t *identity,
4                                   unsigned key_lifetime,
5                                   unsigned flags)
```

The initiator then sends a VERSIONS cell to the responder, which then replies with a VERSIONS cell; they have then negotiated a Tor protocol version. Assuming that the version they negotiate is 3 or higher (the only ones specified for use with this handshake right now), the responder sends a CERTS cell, an AUTH_CHALLENGE cell, and a NETINFO cell to the initiator, which may send either CERTS, AUTHENTICATE, NETINFO if it wants to authenticate, or just NETINFO if it does not. Here is the implementation code.

For backward compatibility between later handshakes and "certificates up-front", the ClientHello of an initiator that supports a later handshake MUST include at least one ciphersuite other than those listed above. The connection responder examines the initiator's ciphersuite list to see whether it includes any ciphers other than those included in the list above. If extra ciphers are included, the responder proceeds as in "renegotiation" and "in-protocol": it sends a single certificate and does not request client certificates. Otherwise (in the case that no extra ciphersuites are included in the ClientHello) the responder proceeds as in "certificates up-front": it requests client certificates, and sends a two-certificate chain. In either case, once the responder has sent its certificate or certificates, the initiator counts them. If two certificates have been sent, it proceeds as in "certificates up-front"; otherwise, it proceeds as in "renegotiation" or "in-protocol".
To decide whether to do "renegotiation" or "in-protocol", the initiator checks whether the responder's initial certificate matches the criteria listed above3.2.

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

Anonymity inside TLS

In all of the above handshake variants, certificates sent in the clear SHOULD NOT include any strings to identify the host as a Tor relay. In the "renegotiation" and "backwards-compatible renegotiation" steps, **the initiator SHOULD choose a list of ciphersuites and TLS extensions to mimic one used by a popular web browser**. Connection protocols are identical for both OP and OR nodes. Onion proxies SHOULD NOT provide long-term-trackable identifiers in their handshakes.

4. Cell Packet Format

Cells

Cell is the basic unit for communication in onion routers and onion proxies. Cells have fixed-width size of **512 bytes**, all of them are consisted by *header* and *payload*.

4.1 Cell versions

Different versions of tor connection have different cell structure:

Table 4.1: Version 1 cell packet format

Command	Length	Description
CircID	CIRCID_LEN bytes	Determines which circuit (if any) the connection is associated with
Command	1 byte	Command field tells what to do with the cell
Payload (padded with padding bytes)	PAYLOAD_LEN bytes	interpretation is based on the command field

Version 2 and higher
All cells have the same format as *Version 1* except the variable length connections.

4.1.1 Headers and Commands

CircID Header

Table 4.2: Version 2 and higher cell format

Command	Length	Description
CircID	CIRCID_LEN bytes	Determines which circuit (if any) the connection is associated with
Command	1 byte	Command field tells what to do with the cell
Payload (padded with padding bytes)	PAYLOAD_LEN bytes	interpretation on the command field
Length	2 bytes big-endian integer	Determines the length of cell it will be specified for processing with a command byte (explained blow)
Payload (some commands MAY pad)	*Length* bytes	interpretation is based on the command field, some commands MAY be padded

The header includes a circuit identifier (*circID*) that specifies which circuit the cell refers to (many circuits can be multiplexed over the single TLS connection), and a command to describe what to do with the cell's payload. (Circuit identifiers are connection-specific: each circuit has a different *circID* on each OP/OR or OR/OR connection it traverses.)

Based on their command, cells are either **control cells**, which are always interpreted by the node that receives them, or **relay cells**, which carry end-to-end stream data.

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

Control cells

The control cell commands are: *padding* (currently used for keepalive, but also usable for link padding); *create* or *created* (used to set up a new circuit); and *destroy* (to tear down a circuit).

2	1	509 bytes
CircId	CMD	DATA

Relay cells

Relay cells have an additional header (the relay header) at the front of the payload, containing a *streamID* (stream identifier: many streams can be multiplexed over a circuit); an end-to-end *checksum* for integrity checking; the *length* of the relay payload; and a relay *command*. The entire contents of the relay header and the relay cell payload are encrypted or decrypted together as the relay cell moves along the circuit, using the **128-bit AES** cipher in counter mode to generate a cipher stream.

Relay commands

The relay commands are: relay *data* (for data flowing down the stream), relay *begin* (to open a stream), relay *end* (to close a stream cleanly), relay *teardown* (to close a broken stream), relay *connected* (to notify the OP that a relay begin has succeeded), relay *extend* and relay *extended* (to

Table 4.3: Type of commands

Value	Command (fixed-length)	Description and References
0	PADDING	padding cell (Sec 7.2)
1	CREATE	Create a circuit (Sec 5.1)
2	CREATED	Acknowledge create (Sec 5.1)
3	RELAY	End-to-end data (Sec 5.5 and Sec 6)
4	DESTROY	destroy a circuit (Sec 5.4)
5	CREATE_FAST	Create a circuit, no PK (Sec 5.1)
6	CREATED_FAST	Circuit created, no PK (Sec 5.1)
8	NETINFO	Time and address info (Sec 4.5)
9	RELAY_EARLY	End-to-end data (Sec 5.6)
10	CREATE2	Extended CREATE cell (Sec 5.1)
11	CREATED2	Extended CREATED cell (Sec 5.1)
12	PADDING_NEGOTIATE	Padding negotiation (Sec 7.2)
Command (variable-length)		
7	VERSIONS	Negotiate proto version (Sec 4)
128	VPADDING	Variable-length padding (Sec 7.2)
129	CERTS	Certificates (Sec 4.2)
130	AUTH_CHALLENGE	Challenge value (Sec 4.3)
131	AUTHENTICATE	Client authentication (Sec 4.5)
132	AUTHORIZE	Client authorization (Reserved for future use)

extend the circuit by a hop, and to acknowledge), relay *truncate* and relay *truncated* (to tear down only part of the circuit, and to acknowledge), relay *sendme* (used for congestion control), and relay *drop* (used to implement long-range dummies).

2	1	2	6	2	1	498 bytes
CircId	Relay	StreamID	Digest	Length	CMD	DATA

4.1.2 Further explanations

Version cells

Most variable-length cells MAY be padded with padding bytes, except for *VERSIONS* cells, which MUST NOT contain any additional bytes. (The payload of *VPADDING* cells consists of padding bytes.) *version* cell is used in negotiation phase.
On a **version 2** connection, variable-length cells are indicated by a command byte equal to 7 ("*VERSIONS*"). On a **version 3 or higher** connection, variable-length cells are indicated by a command byte equal to 7 ("*VERSIONS*"), or greater than or equal to 128.

CircID

CIRCID_LEN is 2 for link protocol versions 1, 2, and 3. CIRCID_LEN is 4 for link protocol version 4 or higher. The first VERSIONS cell, and any cells sent before the first VERSIONS cell, always have CIRCID_LEN == 2 for backward compatibility. The CircID field determines

which circuit, if any, the cell is associated with.

Interpretation of payload

The interpretation of 'Payload' depends on the type of the cell.

VPADDING/PADDING: Payload contains padding bytes.
CREATE/CREATE2: Payload contains the handshake challenge.
CREATED/CREATED2: Payload contains the handshake response.
RELAY/RELAY_EARLY: Payload contains the relay header and relay body.
DESTROY: Payload contains a reason for closing the circuit. (see 5.4)
Upon receiving any other value for the command field, an OR must drop the cell. Since more cell types may be added in the future, ORs should generally not warn when encountering unrecognized commands.

Padding and padding bytes

The cell is padded up to the cell length with padding bytes and receivers MUST ignore padding bytes.
Senders set padding bytes depending on the cell's command:

VERSIONS: Payload MUST NOT contain padding bytes.
AUTHORIZE: Payload is unspecified and reserved for future use.

Other **variable-length** cells:

Payload MAY contain padding bytes at the end of the cell. Padding bytes SHOULD be set to NULL.
RELAY/RELAY_EARLY: Payload MUST be padded to PAYLOAD_LEN with padding bytes. Padding bytes SHOULD be set to random values.

Other **fixed-length** cells:

Payload MUST be padded to PAYLOAD_LEN with padding bytes. Padding bytes SHOULD be set to NULL.
We recommend random padding in *RELAY/RELAY_EARLY* cells, so that the cell content is unpredictable. **See proposal 289 for details** (in short, the traffic can not be easily identified).
For other cells, TLS authenticates cell content, so randomized padding bytes are redundant.

Other commands explanation

PADDING cells are currently used to implement connection keepalive. If there is no other traffic, ORs and OPs send one another a PADDING cell every few minutes.
CREATE, CREATE2, CREATED, CREATED2, and DESTROY cells are used to manage circuits; (see section 5 below) *RELAY* cells are used to send commands and data along a circuit; (see section 6 below)
VERSIONS and NETINFO cells are used to set up connections in link protocols v2 and higher; in link protocol v3 and higher, *CERTS, AUTH_CHALLENGE*, and *AUTHENTICATE* may also be

used. (See section 4 below)

5. Negotiating and initializing connections

After Tor instances negotiate handshake with either the "renegotiation"(v2) or "in-protocol"(v3) handshakes, they must exchange a number of cells to set up the Tor connection and make it open and usable for circuits.

When the **renegotiation** handshake is used,

- Both parties immediately send a VERSIONS cell
- A link protocol version is negotiated (which will be 2)
- Each send a NETINFO cell to confirm their addresses and timestamps.

No other intervening cell types are allowed.

When the **in-protocol** handshake is used,

- the initiator sends a VERSIONS cell to indicate that it will not be renegotiating.
- The responder sends
  - a VERSIONS cell, a CERTS cell 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.
- As soon as it gets the CERTS cell, the initiator knows whether the responder is correctly authenticated. At this point the initiator behaves differently depending on whether it wants to authenticate or not.
  - If it does not want to authenticate, it MUST send a NETINFO cell.
  - If it does want to authenticate, it MUST send a CERTS cell, an AUTHENTICATE cell, and 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.

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

5.1 Negotiating versions with VERSIONS cells

There are multiple instances of the Tor link connection protocol.

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

To determine the version, in any connection where the "renegotiation" or "in-protocol" handshake was used (that is, where the responder sent only one certificate at first and where the initiator did not send any certificates in the first negotiation), both parties MUST send a VERSIONS cell.

- In "renegotiation", they send a VERSIONS cell right after the renegotiation is finished, before any other cells are sent.
- In "in-protocol", the initiator sends a VERSIONS cell immediately after the initial TLS handshake, and the responder replies immediately with a VERSIONS cell. (As an exception to this rule, if both sides support the "in-protocol" handshake, either side may send VPADDING cells at any time.)

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

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.)

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

Link protocols differences are:

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

5.2 CERTS cells

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

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

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

Relevant certType values are:

1. Link key certificate certified by RSA1024 identity
2. RSA1024 Identity certificate, self-signed.2
3. RSA1024 AUTHENTICATE cell link certificate, signed with RSA1024 key.
4. Ed25519 signing key, signed with identity key.2
5. TLS link certificate, signed with ed25519 signing key.
6. Ed25519 AUTHENTICATE cell key, signed with ed25519 signing key.
7. Ed25519 identity, signed with RSA identity.

The certificate format for certificate types 1-3 is DER encoded X509. For types 4-6 ED25519 encoding is used. For type 7 RSA CrossCert encoding is used.

X509 certificated encoding

In cryptography, X.509 is a standard defining the format of public key certificates. X.509 certificates are used in many Internet protocols, including TLS/SSL, which is the basis for HTTPS, the secure protocol for browsing the web. They are also used in offline applications, like electronic signatures. An X.509 certificate contains a public key and an identity (a hostname, or an organization, or an individual), and is either signed by a certificate authority or self-signed. When a certificate is signed by a trusted certificate authority, or validated by other means, someone holding that certificate can rely on the public key it contains to establish secure communications with another party, or validate documents digitally signed by the corresponding private key.

ED25519 encoding

This encoding has the following structure:

```
struct ed25519_cert_st {
    uint8_t version;
    uint8_t cert_type;
    uint32_t exp_field;
    uint8_t cert_key_type;
    uint8_t certified_key[32];
    uint8_t n_extensions;
    TRUNNEL_DYNARRAY_HEAD(, struct ed25519_cert_extension_st *) ext;
    uint8_t signature[64];
    uint8_t trunnel_error_code_;
};
```

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

To authenticate **the responder**, **the initiator** must check a set of conditions , based on which of these cases has happened:

1. having a given Ed25519, RSA identity key combination
    - The CERTS cell contains exactly one CertType 2 "ID" certificate.
    - The CERTS cell contains exactly one CertType 4 Ed25519 "Id->Signing" cert.
    - The CERTS cell contains exactly one CertType 5 Ed25519 "Signing->link" certificate.
    - The CERTS cell contains exactly one CertType 7 "RSA->Ed25519" cross-certificate.
    - All X.509 certificates above have validAfter and validUntil dates; no X.509 or Ed25519 certificates are expired.
    - All certificates are correctly signed.
    - The certified key in the Signing->Link certificate matches the SHA256 digest of the certificate that was used to authenticate the TLS connection.
    - The identity key listed in the ID->Signing cert was used to sign the ID->Signing Cert.
    - The Signing->Link cert was signed with the Signing key listed in the ID->Signing cert.
    - The RSA->Ed25519 cross-certificate certifies the Ed25519 identity, and is signed with the RSA identity listed in the "ID" certificate.
    - The certified key in the ID certificate is a 1024-bit RSA key.
    - The RSA ID certificate is correctly self-signed.
2. having a given RSA identity only
    - The CERTS cell contains exactly one CertType 1 "Link" certificate.
    - The CERTS cell contains exactly one CertType 2 "ID" certificate.
    - Both certificates have validAfter and validUntil dates that are not expired.
    - The certified key in the Link certificate matches the link key that was used to negotiate the TLS connection.
    - The certified key in the ID certificate is a 1024-bit RSA key.
    - The certified key in the ID certificate was used to sign both certificates.
    - The link certificate is correctly signed with the key in the ID certificate
    - The ID certificate is correctly self-signed

In both cases above, checking these conditions is sufficient to authenticate that the initiator is talking to the Tor node with the expected identity, as certified in the ID certificate(s).

To authenticate **the initiator**, **the responder** must check a set of conditions , based on which of these cases has happened:

1. having a given Ed25519, RSA identity key combination
    - The CERTS cell contains exactly one CertType 2 "ID" certificate.
    - The CERTS cell contains exactly one CertType 4 Ed25519 "Id->Signing" certificate.
    - The CERTS cell contains exactly one CertType 6 Ed25519 "Signing->auth" certificate.
    - The CERTS cell contains exactly one CertType 7 "RSA->Ed25519" cross-certificate.
    - All X.509 certificates above have validAfter and validUntil dates; no X.509 or Ed25519 certificates are expired.
    - All certificates are correctly signed.
    - The identity key listed in the ID->Signing cert was used to sign the ID->Signing Cert.
    - The Signing->AUTH cert was signed with the Signing key listed in the ID->Signing

cert.
- The RSA->Ed25519 cross-certificate certifies the Ed25519 identity, and is signed with the RSA identity listed in the "ID" certificate.
- The certified key in the ID certificate is a 1024-bit RSA key.
- The RSA ID certificate is correctly self-signed.

2. having a given RSA identity only
   - The CERTS cell contains exactly one CertType 3 "AUTH" certificate.
   - The CERTS cell contains exactly one CertType 2 "ID" certificate.
   - Both certificates have validAfter and validUntil dates that are not expired.
   - The certified key in the AUTH certificate is a 1024-bit RSA key.
   - The certified key in the ID certificate is a 1024-bit RSA key.
   - The certified key in the ID certificate was used to sign both certificates.
   - The auth certificate is correctly signed with the key in the ID certificate.
   - The ID certificate is correctly self-signed.

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

5.3 AUTH_CHALLENGE cells

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

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

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

The Challenge field is a randomly generated string that the initiator must sign (a hash of) as part of authenticating. The methods are the authentication methods that the responder will accept. Only two authentication methods are defined right now: RSA-SHA256-TLSSecret and Ed25519-SHA256-RFC5705.

5.4 AUTHENTICATE cells

If an initiator wants to authenticate, it responds to the AUTH_CHALLENGE cell with a CERTS cell and an AUTHENTICATE cell. The CERTS cell is as a server would send, except that instead of sending a CertType 1 (and possibly CertType 5) certs for arbitrary link certificates, the initiator sends a CertType 3 (and possibly CertType 6) cert for an RSA/Ed25519 AUTHENTICATE key. This difference is because we allow any link key type on a TLS link, but the protocol described here will only work for specific key types as described below.

An AUTHENTICATE cell contains the following:

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

Responders MUST ignore extra bytes at the end of an AUTHENTICATE cell. Recognized AuthTypes are 1 and 3, described in the next two sections.

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

5.4.1 Link authentication type 1: RSA-SHA256-TLSSecret

If AuthType is 1 (meaning "RSA-SHA256-TLSSecret"), then the Authentication field of the AUTHENTICATE cell contains the following:

- TYPE: The characters "AUTH0001" [8 octets]
- CID: A SHA256 hash of the initiator's RSA1024 identity key [32 octets]
- SID: A SHA256 hash of the responder's RSA1024 identity key [32 octets]
- SLOG: A SHA256 hash of all bytes sent from the responder to the initiator as part of the negotiation up to and including the AUTH_CHALLENGE cell; that is, the VERSIONS cell, the CERTS cell, the AUTH_CHALLENGE cell, and any padding cells. [32 octets]
- CLOG: A SHA256 hash of all bytes sent from the initiator to the responder as part of the negotiation so far; that is, the VERSIONS cell and the CERTS cell and any padding cells. [32 octets]
- SCERT: A SHA256 hash of the responder's TLS link certificate. [32 octets]
- TLSSECRETS: A SHA256 HMAC, using the TLS master secret as the secret key, of the following:
  - client_random, as sent in the TLS Client Hello
  - server_random, as sent in the TLS Server Hello
  - the NUL terminated ASCII string:
    "Tor V3 handshake TLS cross-certification"

  [32 octets]
- RAND: A 24 byte value, randomly chosen by the initiator. (In an imitation of SSL3's gmt_unix_time field, older versions of Tor sent an 8-byte timestamp as the first 8 bytes of this field; new implementations should not do that.) [24 octets]
- SIG: A signature of a SHA256 hash of all the previous fields using the initiator's "Authenticate" key as presented. (As always in Tor, we use OAEP-MGF1 padding; see tor-spec.txt section 0.3.) [variable length]

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

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

5.4.2 Link authentication type 3: Ed25519-SHA256-RFC5705

If AuthType is 3, meaning "Ed25519-SHA256-RFC5705", the Authentication field of the AuthType cell is as below:

Modified values and new fields below are marked with asterisks.

- TYPE: The characters "AUTH0003" [8 octets]
- CID: A SHA256 hash of the initiator's RSA1024 identity key [32 octets]
- SID: A SHA256 hash of the responder's RSA1024 identity key [32 octets]
- CID_ED: The initiator's Ed25519 identity key [32 octets]
- SID_ED: The responder's Ed25519 identity key, or all-zero. [32 octets]
- SLOG: A SHA256 hash of all bytes sent from the responder to the initiator as part of the negotiation up to and including the AUTH_CHALLENGE cell; that is, the VERSIONS cell, the CERTS cell, the AUTH_CHALLENGE cell, and any padding cells. [32 octets]
- CLOG: A SHA256 hash of all bytes sent from the initiator to the responder as part of the negotiation so far; that is, the VERSIONS cell and the CERTS cell and any padding cells. [32 octets]
- SCERT: A SHA256 hash of the responder's TLS link certificate. [32 octets]
- TLSSECRETS: The output of an RFC5705 Exporter function on the TLS session, using as its inputs:
  - The label string "EXPORTER FOR TOR TLS CLIENT BINDING AUTH0003"
  - The context value equal to the initiator's Ed25519 identity key.
  - The length 32.

  [32 octets]
- RAND: A 24 byte value, randomly chosen by the initiator. [24 octets]
- SIG: A signature of all previous fields using the initiator's Ed25519 authentication key (as in the cert with CertType 6). [variable length]

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

5.5 NETINFO cells

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

```
TIME (Timestamp) [4 bytes]
OTHERADDR (Other OR's address) [variable]
    ATYPE (Address type) [1 byte]
    ALEN (Adress length) [1 byte]
    AVAL (Address value in NBO) [ALEN bytes]
NMYADDR (Number of this OR's addresses) [1 byte]
NMYADDR times:
    ATYPE (Address type) [1 byte]
    ALEN (Adress length) [1 byte]
    AVAL (Address value in NBO)) [ALEN bytes]
```

Recognized address types (ATYPE) are:

```
[04] IPv4.
[06] IPv6.
```

ALEN MUST be 4 when ATYPE is 0x04 (IPv4) and 16 when ATYPE is 0x06 (IPv6).

The timestamp is a big-endian unsigned integer number of seconds since the Unix epoch. Implementations MUST ignore unexpected bytes at the end of the cell.

Implementations MAY use the timestamp value to help decide if their clocks are skewed. Initiators MAY use "other OR's address" to help learn which address their connections may be originating from, if they do not know it; and to learn whether the peer will treat the current connection as canonical. Implementations SHOULD NOT trust these values unconditionally, especially when they come from non-authorities, since the other party can lie about the time or IP addresses it sees.

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

6. Circuit management

6.1 CREATE and CREATED cells

Users set up circuits incrementally, one hop at a time. To create a new circuit, OPs send a CREATE/CREATE2 cell to the first node, with the first half of an authenticated handshake; that node responds with a CREATED/CREATED2 cell with the second half of the handshake. To extend a circuit past the first hop, the OP sends an EXTEND/EXTEND2 relay cell, which instructs the last node in the circuit to send a CREATE/CREATE2 cell to extend the circuit.

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

A CREATE2 cell contains:

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

A CREATED2 cell contains:

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

Recognized handshake types are:

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

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

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

or

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

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

```
ntor -- 'ntorNTORntorNTOR'
```

The format of a CREATED cell is:

```
HDATA (Server Handshake Data) [TAP_S_HANDSHAKE_LEN bytes]
```

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

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

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

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

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

6.1.1 Handling CREATED Cells

When CREATED/CREATED2 cells are parsed and checked, they are handed off to a CPUWorker as a task. If the queue is full, tasks will be added to the pending queue. Otherwise, a cpuworker_request_t will be build with the received handshake data. A cpuworker_job_t will then be created containing the cpuworker request.

The job will further be encapsulated in a workqueue_entry_t instance and will be added to queue of entries. Whenever a thread becomes idle, it will dequeue this entry and call on the request the cpuworker_onion_handshake_threadfn function, which will generate the handshake answer based on the on its type, and will save it in the cpuworker_reply_t field of the job. The reply will be added to reply_queue. Reply's type will be CREATED/CREATED2/CREATED_FAST.

the cpuworker_onion_handshake_replyfn function will later be called upon the reply, which will append the CREATED cell to the circuit queue, to be sent on the network to the initiating node.

6.1.2 Choosing circuit IDs in create cells

The CircID for a CREATE/CREATE2 cell is an arbitrarily chosen nonzero integer, selected by the node (OP or OR) that sends the CREATE/CREATE2 cell. In link protocol 3 or lower, CircIDs are 2 bytes long; in protocol 4 or higher, CircIDs are 4 bytes long.

To prevent CircID collisions, when one node sends a CREATE/CREATE2 cell to another, it chooses from only one half of the possible values based on the ORs' public identity keys.

In link protocol version 3 or lower, if the sending node has a lower key, it chooses a CircID with an MSB of 0; otherwise, it chooses a CircID with an MSB of 1. (Public keys are compared numerically by modulus.) With protocol version 3 or lower, a client with no public key MAY choose any CircID it wishes, since clients never need to process a CREATE/CREATE2 cell.

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

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.

6.1.3 EXTEND and EXTENDED cells

To extend an existing circuit, the client sends an EXTEND or EXTEND2 relay cell to the last node in the circuit. An EXTEND2 cell's relay payload contains:

```
NSPEC (Number of link specifiers) [1 byte]
NSPEC times:
    LSTYPE (Link specifier type) [1 byte]
    LSLEN (Link specifier length) [1 byte]
    LSPEC (Link specifier) [LSLEN bytes]
HTYPE (Client Handshake Type) [2 bytes]
HLEN (Client Handshake Data Len) [2 bytes]
HDATA (Client Handshake Data) [HLEN bytes]
```

Link specifiers describe the next node in the circuit and how to connect to it. Recognized specifiers are:

00 TLS-over-TCP, IPv4 address A four-byte IPv4 address plus two-byte ORPort
01 TLS-over-TCP, IPv6 address A sixteen-byte IPv6 address plus two-byte ORPort
02 Legacy identity A 20-byte SHA1 identity fingerprint. At most one may be listed.
03 Ed25519 identity A 32-byte Ed25519 identity fingerprint. At most one may be listed.

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

The relay payload for an EXTEND relay cell consists of:

```
Address [4 bytes]
Port [2 bytes]
Onion skin [TAP_C_HANDSHAKE_LEN bytes]
Identity fingerprint [HASH_LEN bytes]
```

The "legacy identity" and "identity fingerprint" fields are the SHA1 hash of the PKCS#1 ASN1 encoding of the next onion router's identity (signing) key. (See 0.3 above.) The "Ed25519 identity" field is the Ed25519 identity key of the target node. Including this key information allows the extending OR verify that it is indeed connected to the correct target OR, and prevents certain man-in-the-middle attacks.

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

If an extending OR 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. After checking relay identities, extending ORs generate a CREATE/CREATE2 cell from the contents of the EXTEND/EXTEND2 cell.

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

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

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

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

When generating an EXTEND2 cell, clients SHOULD include the target's Ed25519 identity whenever the target has one, and whenever the target supports LinkAuth subprotocol version "3".

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

6.1.4 The "TAP" handshake

This handshake uses Diffie-Hellman in Z_p and RSA to compute a set of shared keys which the client knows are shared only with a particular server, and the server knows are shared with

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

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

The payload for a CREATE cell is an 'onion skin', which consists of the first step of the DH handshake data (also known as g^x). This value is encrypted using the "legacy hybrid encryption" algorithm (see 0.4 above) to the server's onion key, giving a client handshake:

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

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

```
DH data (g^y) [DH_LEN bytes]
Derivative key data (KH) [HASH_LEN bytes]
```

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

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

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

6.1.5 The "ntor" handshake

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

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

In this section, define:

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

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

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

and generates a client-side handshake with contents:

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

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

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

The server's handshake reply is:

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

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

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

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

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

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

6.1.6 CREATE_FAST/CREATED_FAST cells

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

A CREATE_FAST cell contains:

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

A CREATED_FAST cell contains:

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

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.

The CREATE_FAST handshake is currently deprecated whenever it is not necessary;

[Tor 0.3.1.1-alpha and later disable CREATE_FAST by default.]

6.2 Setting circuit keys

6.2.1 KDF-TOR

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

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

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

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

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

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

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

6.2.2 KDF-RFC5869

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

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

Where

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

In RFC5869's vocabulary, this is HKDF-SHA256 with

```
info == m_expand,
salt == t_key,
IKM == secret_input.
```

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

6.3 Creating circuits

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

1. Choose an onion router as an 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 6.1 . Some end nodes accept streams, others are introduction or rendezvous points (see rend-spec-v2,v3.txt).

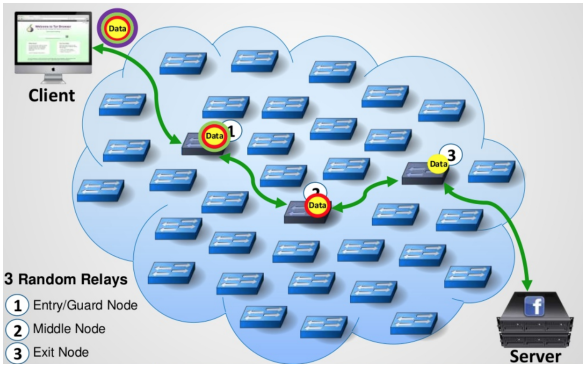


Figure 6.1: Tor Circuit Creation [2]

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 OP performs these steps:

1. Create an onion skin, encrypted to R_M's public onion key.
2. Send the onion skin in a relay EXTEND/EXTEND2 cell along the circuit (see sections 5.1.2 and 5.5).
3. When a relay EXTENDED/EXTENDED2 cell is received, verify KH, and calculate the shared keys. The circuit is now extended. When an onion router receives an EXTEND relay cell, it sends a CREATE cell to the next onion router, with the enclosed onion skin as its payload.

When an onion router receives an EXTEND2 relay cell, it sends a CREATE2 cell to the next onion router, with the enclosed HLEN, HTYPE, and HDATA as its payload.

As special cases, if the EXTEND/EXTEND2 cell includes a legacy identity, identity fingerprint, or Ed25519 identity of all zeroes, or asks to extend back to the relay that sent the extend cell, the circuit will fail and be torn down. The initiating onion router chooses some circID not yet used on the connection between the two onion routers.

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 it payload into an EXTENDED/EXTENDED2 relay cell, and sends that cell up the circuit. Upon receiving the EXTENDED/EXTENDED2 relay cell, the OP can retrieve the handshake material.

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

6.3.1 Canonical connections

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

To prevent this, when an OR gets an extend request, it SHOULD use an existing OR connection if the ID matches, and ANY of the following conditions hold:

- The IP matches the requested IP.
- The OR knows that the IP of the connection it's using is canonical because it was listed in the NETINFO cell.
- The OR knows that the IP of the connection it's using is canonical because it was listed in the server descriptor.

6.4 Tearing down circuits

Circuits are torn down when an unrecoverable error occurs along the circuit, or when all streams on a circuit are closed and the circuit's intended lifetime is over.

ORs 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.

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

- If an OR sees a circuit created with CREATE_FAST, the OR is sure to be the first hop of a circuit.
- If an OR is the responder, and the initiator:
  - did not authenticate the link, or
  - authenticated with a key that is not in the consensus, then the OR 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, an OR or OP sends a DESTROY cell to the adjacent nodes on that circuit, using the appropriate direction's circID.

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

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

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

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

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

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

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

13. NOSUCHSERVICE (Request for unknown hidden service)

6.5 Routing relay cells

6.5.1 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.

6.5.2 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 an OP, the OP encrypts the payload with the stream cipher as follows:

```
OP 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 an OR, it decrypts the payload with the stream cipher, as follows:

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

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

For more information, see section 6 below.

6.5.3 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 an OR, it encrypts the payload with the stream cipher, as follows:

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

Routing to the Origin

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

```
OP receives relay cell from node 1:
    For I=1...N, where N is the final node on the circuit:
```

6.5.4 Handling relay_early cells

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

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

When speaking v2 of the link protocol or later, clients MUST only send EXTEND/EXTEND2 cells inside RELAY_EARLY cells. Clients SHOULD send the first 8 RELAY cells that are not targeted at the first hop of any circuit as RELAY_EARLY cells too, in order to partially conceal the circuit length.

[Starting with Tor 0.2.3.11-alpha, relays should reject any EXTEND/EXTEND2 cell not received in a RELAY_EARLY cell.]

7. Application connections and stream managem

7.1 Relay cells

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

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 payload of each unencrypted RELAY cell consists of:

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

The relay commands are:

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

```
7 -- RELAY_EXTENDED [backward] [control]
8 -- RELAY_TRUNCATE [forward] [control]
9 -- RELAY_TRUNCATED [backward] [control]
10 -- RELAY_DROP [forward or backward] [control]
11 -- RELAY_RESOLVE [forward]
12 -- RELAY_RESOLVED [backward]
13 -- RELAY_BEGIN_DIR [forward]
14 -- RELAY_EXTEND2 [forward] [control]
15 -- RELAY_EXTENDED2 [backward] [control]
32..40 -- Used for hidden services; see rend-spec-{v2,v3}.txt.
```

Commands labelled as "forward" must only be sent by the originator of the circuit. Commands labelled as "backward" must only be sent by other nodes in the circuit back to the originator. Commands marked as either can be sent either by the originator or other nodes. The 'recognized' field 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 payload 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, and including this RELAY cell's entire payload (taken with the digest field set to zero). If the digest is correct, the cell is considered "recognized" for the purposes of decryption.

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

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

The 'Length' field of a relay cell contains the number of bytes in the relay payload which contain real payload data. The remainder of the unencrypted payload is padded with padding bytes. Implementations handle padding bytes of unencrypted relay cells as they do padding bytes for other cell types;

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.

7.2 Opening streams and transferring data

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

```
ADDRPORT [nul-terminated string]
    ADDRESS | ':' | PORT | [00]
FLAGS [4 bytes]
```

where ADDRESS can be a

- DNS hostname;
- IPv4 address in dotted-quad format;
- IPv6 address surrounded by square brackets.

And where PORT is a decimal integer between 1 and 65535, inclusive.

The FLAGS value has one or more of the following bits set, where "bit 01" is the LSB of the 32-bit value, and "bit 32" is the MSB. (Remember that all values 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.)

bit meaning:

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

Upon receiving this cell, the exit node resolves the address as necessary, and opens a new TCP connection to the target port. If the address cannot be resolved, or a connection can't be established, the exit node replies with a RELAY_END cell.

Otherwise, the exit node replies with a RELAY_CONNECTED cell, whose payload is in one of the following formats:

```
The IPv4 address to which the connection was made [4 octets]
A number of seconds (TTL) for which the address may be cached [4 octets]
```

or

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

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

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

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

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

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

7.2.1 Opening a directory stream

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

Directory servers may be:

- authoritative directories (RELAY_BEGIN_DIR, usually non-anonymous),
- bridge authoritative directories (RELAY_BEGIN_DIR, anonymous),
- directory mirrors (RELAY_BEGIN_DIR, usually non-anonymous),
- onion service directories (RELAY_BEGIN_DIR, anonymous).

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

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

In response to a RELAY_BEGIN_DIR cell, relays respond either with a RELAY_CONNECTED cell on succcess, or a RELAY_END cell on failure. They MUST send a RELAY_CONNECTED cell all-zero payload, and clients MUST ignore the payload.

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

7.3 Closing streams

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

The payload of a RELAY_END cell begins with a single 'reason' byte to describe why the stream is closing. For some reasons, it contains additional data (depending on the reason.) The values are:

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

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

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

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

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

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

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

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

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

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

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

7.4 Remote hostname lookup

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

```
Type (1 octet)
Length (1 octet) that is the length of the Value field
Value (variable-width)
TTL (4 octets)
```

"Type" is one of:

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

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

The 'Value' field encodes the answer: IP addresses are given in network order. Hostnames are given in standard DNS order ("www.example.com") and not NUL-terminated.

The content of Errors is currently ignored. Relays currently set it to the string "Error resolving hostname" with no terminating NUL. Implementations MUST ignore this value.

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

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

8. Rate Limiting And Fairness

Intro

Not all the users of *Tor* wish to service as much as they can, they maybe want to limit their service and bandwidth usage. In this chapter we will investigate the Tors approach toward rate limiting and fairness.

Tors Approach

Tor takes the *token bucket* approach (see Fig 8.1) for achieving rate limiting to the users. with this approach, the average rate of output will be equal to incoming bytes while allowing short-term burst. so with this approach in practice the incoming bytes to be transferred will be limited.

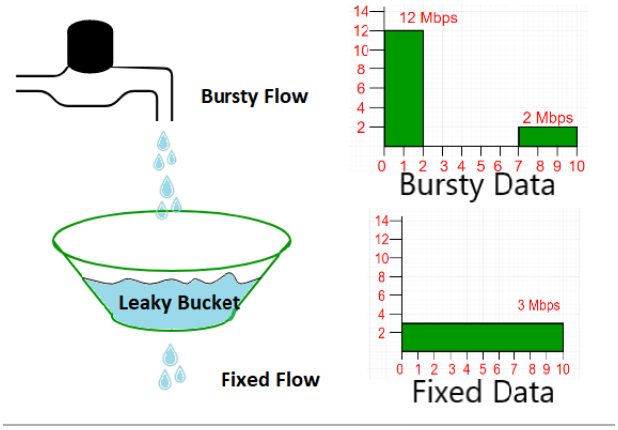


Figure 8.1: Token Bucket Approach - from *geekforgeeks.com* website

Distinguishing Interactive and Bulk Streams

have in mind that for TCP connections, for every bit of information to transfer we have to transfer a whole cell of 512 bytes (we can't wait to receive enough bytes to fill-up the cell, maybe it is an interactive scenario that the sender waits for the reply).
In-order to better service the clients, Tor distinguishes between bulk streams and interactive streams by their frequency in which they supply cells. the algorithm and idea is based on Rennhard et al's design. with this approach Tor tries to give a preferential service to interactive streams and good overall service (best effort) to bulk streams. but have in mind with this preferential services the timing end-to-end attack is possible.

Rate Limiting Implementation

Tor, as we mentioned before, uses a token bucket approach (one for reads, one for writes).
Since version 0.2.0.X tor let the clients use additional token buckets to specify their strict rate limiting policies on the "relayed" traffic.
To avoid partitioning concerns we combine both classes of traffic over a given OR connection, and keep track of the last time we read or wrote a high-priority (non-relayed) cell. If it's been less than N seconds (currently N=30), we give the whole connection high priority, else we give the whole connection low priority. We also give low priority to reads and writes for connections that are serving directory information. see proposal 111 for more details.

9. Congestion and Flow Control

Intro

Rate limiting is not enough Of course and Congestion control mechanism for better servicing will be necessary. imagine many clients with huge amount of traffic use the same circuit, the consequence will be that the circuit might become saturated by a time. another scenario might be that an attacker sends a huge amount of data that do not read it at the end. so here some congestion control and flow control mechanism is needed, we will describe the Tors approach for the problem.

9.1 Tors Approach

Tor will use two methods for solution that the second uses a modified version of first solution. the first solution will discuss about congestion control for circuits and then the second solution will discuss on multiple streams that are going through that circuit.

9.1.1 Circuit-level throttling

To control a circuit's bandwidth usage, each OR keeps track of two windows. (See Fig 9.1) The packaging window tracks how many relay data cells the OR is allowed to package (from incoming TCP streams) for transmission back to the OP, and the delivery window tracks how many relay data cells it is willing to deliver to TCP streams outside the network. Each window is initialized (say, to 1000 data cells).

When a data cell is packaged or delivered, the appropriate window is decremented. When an OR has received enough data cells (currently 100, see dir-spec.txt), it sends a relay sendme cell towards the OP, with streamID zero. When an OR receives a relay sendme cell with streamID zero (The body SHOULD be ignored) , it increments its packaging window. Either of these cells increments the corresponding window by 100. If the packaging window reaches 0, the OR stops reading from

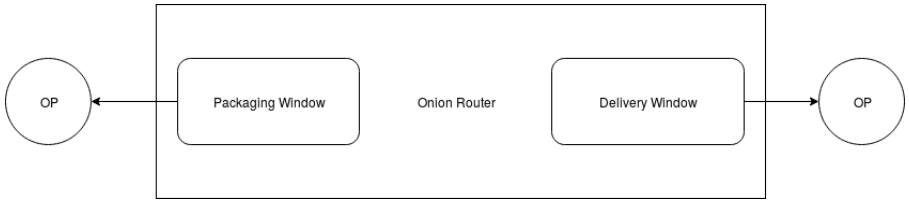


Figure 9.1: The delivery and packaging window in a OR/OP

TCP connections for all streams on the corresponding circuit, and sends no more relay data cells until receiving a relay sendme cell. You can see the sample scenario in Fig 9.2 .
The OP behaves identically, except that it must track a packaging window and a delivery window for every OR in the circuit. If a packaging window reaches 0, it stops reading from streams destined for that OR.
An OR or OP sends cells to increment its delivery window when the corresponding window value falls under some threshold (900).

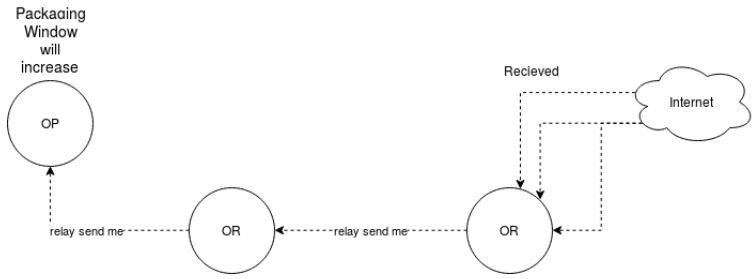
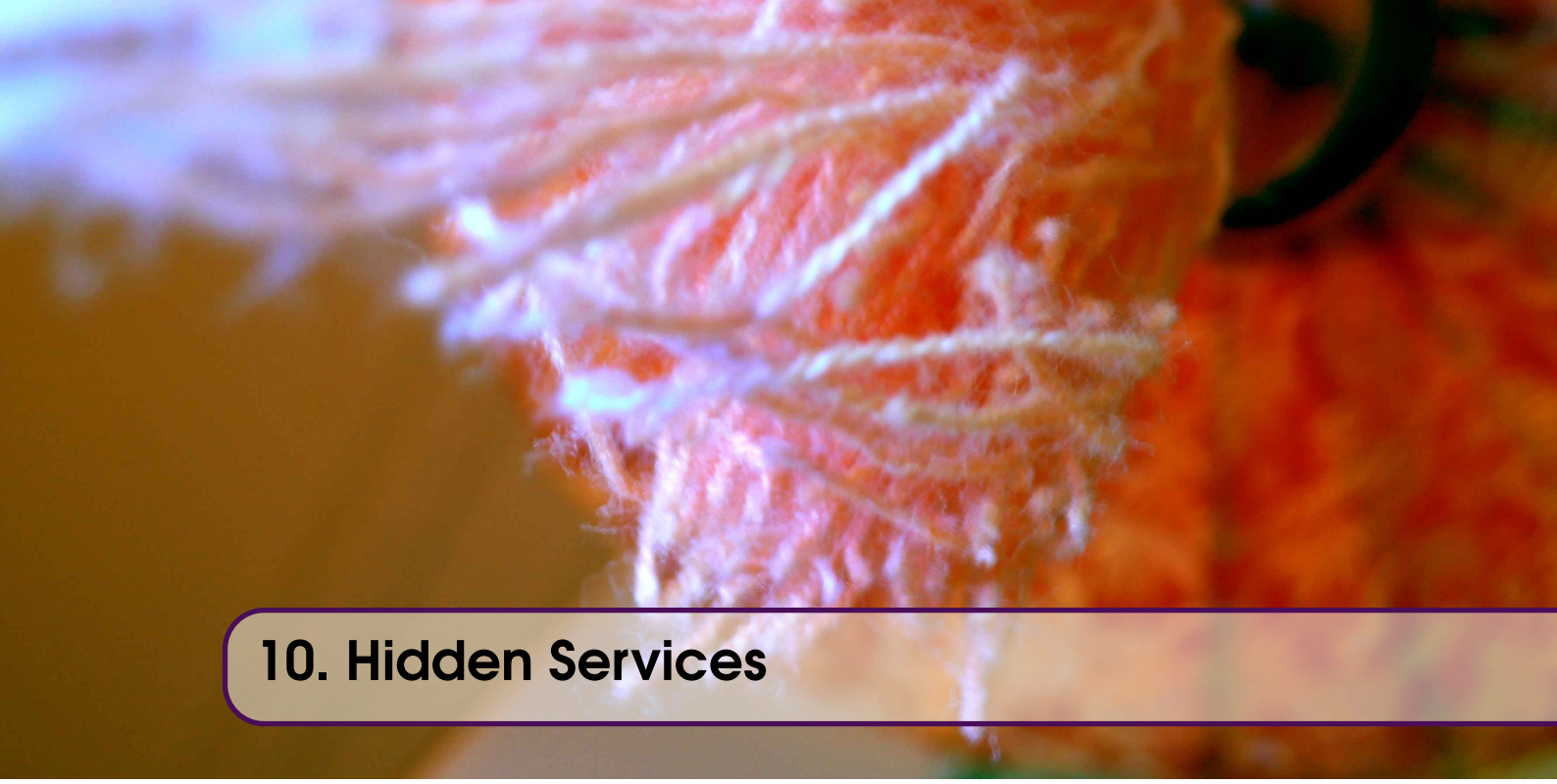


Figure 9.2: Circuit-level Congestion Control Scenario

9.1.2 Stream-level throttling

Stream-level Congestion Control is the same, ORs and OPs use relay sendme cells to implement end-to-end flow control for individual streams across circuits.
Each stream begins with a packaging window (currently 500 cells), and increments the window by a fixed value (50) upon receiving a relay sendme cell.
Rather than always returning a relay sendme cell as soon as enough cells have arrived, the stream level congestion control also has to check whether data has been successfully flushed onto the TCP stream; it sends the relay sendme cell only when the number of bytes pending to be flushed is under some threshold (currently 10 cells worth).
for further information please refer to Section 8 of TorDesign. [4]
Stream-level RELAY_SENDME cells are distinguished by having nonzero StreamID. They are still empty; the body still SHOULD be ignored.

II Network Components



10. Hidden Services

10.1 Rendezvous Points and hidden services

Rendezvous points are a building block for location-hidden services (also known as responder anonymity) in the Tor net- work. Location-hidden services allow Bob to offer a TCP service, such as a webserver, without revealing his IP address.[4]

We provide location-hiding for Bob by allowing him to advertise several onion routers (his introduction points) as contact points.[4] He may do this on any robust efficient key-value lookup system with authenticated updates, such as a distributed hash table (DHT) like CFS.[4] [1] Alice, the client, chooses an OR as her rendezvous point. She connects to one of Bob's introduction points, informs him of her rendezvous point, and then waits for him to connect to the rendezvous point.[4]

10.1.1 Rendezvous points in Tor

The following steps are performed on behalf of Alice and Bob by their local OPs [4] :

- Bob generates a long-term public key[2] pair to identify his service.
- Bob chooses some introduction points, and advertises them on the lookup service, signing the advertisement with his public key. He can add more later.
- Bob builds a circuit to each of his introduction points, and tells them to wait for requests.
- Alice learns about Bob's service out of band (perhaps Bob told her, or she found it on a website). She retrieves the details of Bob's service from the lookup service. If Alice wants to access Bob's service anonymously, she must connect to the lookup service via Tor.
- Alice chooses an OR as the rendezvous point (RP) for her connection to Bob's service. She builds a circuit to the RP, and gives it a randomly chosen "rendezvous cookie" to recognize Bob.
- Alice opens an anonymous stream to one of Bob's introduction points, and gives it a message (encrypted with Bob's public key) telling it about herself, her RP and rendezvous cookie, and

[1] This DHTs is distributed across many Tor relays (HSDirs) Any regular Tor relay may work as HSDir, anyone who deploys such HSDir nodes can also harvest onion addresses from it. The adversary may find addresses that have not been publicly shared ever. [2]

[2] The hostname is just the (truncated) hash of the public key[6]

the start of a DH handshake. The introduction point sends the message to Bob.

- If Bob wants to talk to Alice, he builds a circuit to Al- ice's RP and sends the rendezvous cookie, the second half of the DH handshake, and a hash of the session key they now share.
- The RP connects Alice's circuit to Bob's. Note that RP can't recognize Alice, Bob, or the data they transmit.
- Alice sends a relay begin cell along the circuit. It arrives at Bob's OP, which connects to Bob's webserver.
- An anonymous stream has been established, and Alice and Bob communicate as normal.

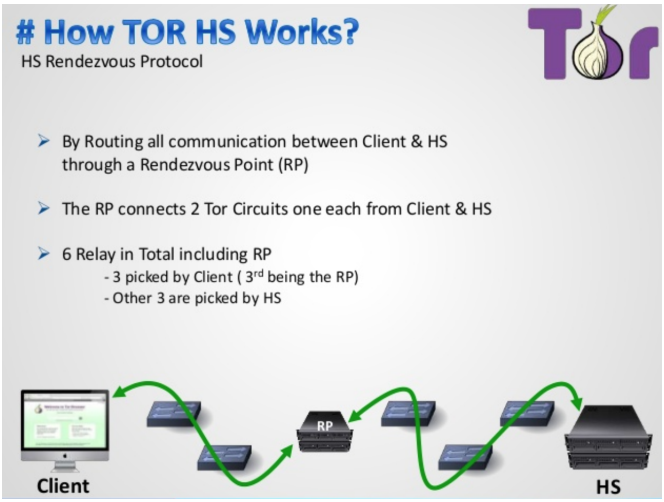


Figure 10.1: Tor Hidden services [2]

Up to at least October 2013 the hidden services work like this

1. A hidden service calculates its key pair (private and public key, asymmetric encryption).
2. Then the hidden service picks some relays as its introduction points.
3. It tells its public key to those introduction points over Tor circuits[3] .
4. After that the hidden-service creates a hidden service descriptor, containing its public key and what its introduction points are.
5. The hidden service signs the hidden service descriptor with its private key.
6. It then uploads the hidden service descriptor to a distributed hash table (DHT).
7. Clients learn the .onion address from a hidden service out-of-band. (e.g. public website) [4]
8. After retrieving the .onion address the client connects to the DHT and asks for that hash.
9. If it exists the client learns about the hidden service's public key and its introduction points.
10. The client picks a relay at random to build a circuit to it, to tell it a one-time secret. The picked relay acts as rendezvous point.
11. The client creates an introduce message, containing the address of the rendezvous point and the one-time secret, before encrypting the message with the hidden service's public key.
12. The client sends its message over a Tor circuit to one of the introduction points, demanding it to be forwarded to the hidden service.
13. The hidden service decrypts the introduce message with its private key to learn about the rendezvous point and the one-time secret.
14. The hidden service creates a rendezvous message, containing the one-time secret and sends it over a circuit to the rendezvous point.

[3] We need to understand message formats for this action

[4] A hash.onion is a 16 character name derived from the service's public key.

15. The rendezvous point tells the client that a connection was established.
16. Client and hidden service talk to each other over this rendezvous point. All traffic is end-to-end encrypted and the rendezvous point just relays it back and forth. Note that each of them, client and hidden service, build a circuit to the rendezvous point; at three hops per circuit this makes six hops in total 10.2 .

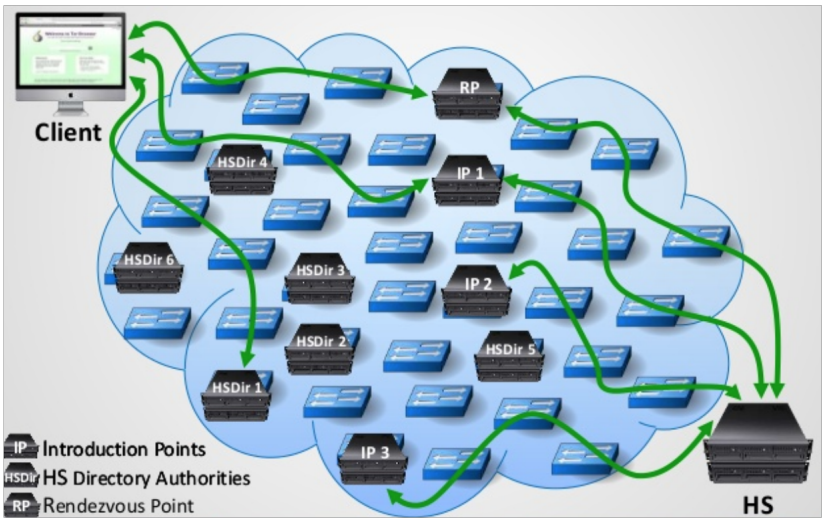


Figure 10.2: Tor Hidden services and HS directories [2]

III Cryptography Guide

10.2 AES

AES is based on a design principle known as a substitution–permutation network, and is efficient in both software and hardware.

High-level description of the algorithm:

1. KeyExpantion: round keys are derived from the cipher key using Rijndael's key schedule. AES requires a separate 128-bit round key block for each round plus one more.
2. Initial round key addition:
   (a) AddRoundKey: each byte of the state is combined with a block of the round key using bitwise xor.
3. For 9, 11 or 13 rounds:
   (a) SubBytes: a non-linear substitution step where each byte is replaced with another according to a lookup table.
   (b) ShiftRows: a transposition step where the last three rows of the state are shifted cyclically a certain number of steps.
   (c) MixColumns: a linear mixing operation which operates on the columns of the state, combining the four bytes in each column.
   (d) AddRoundKey
4. Final round (making 10, 12 or 14 rounds in total):
   (a) SubBytes
   (b) ShiftRows
   (c) AddRoundKey

In cryptography, a block cipher by itself is only suitable for the secure cryptographic transformation (encryption or decryption) of one fixed-length group of bits called a block. A mode of operation describes how to repeatedly apply a cipher's single-block operation to securely transform amounts of data larger than a block.

Counter mode encryption

The figure 10.3 will show a block cipher encryption with counter mode operation.

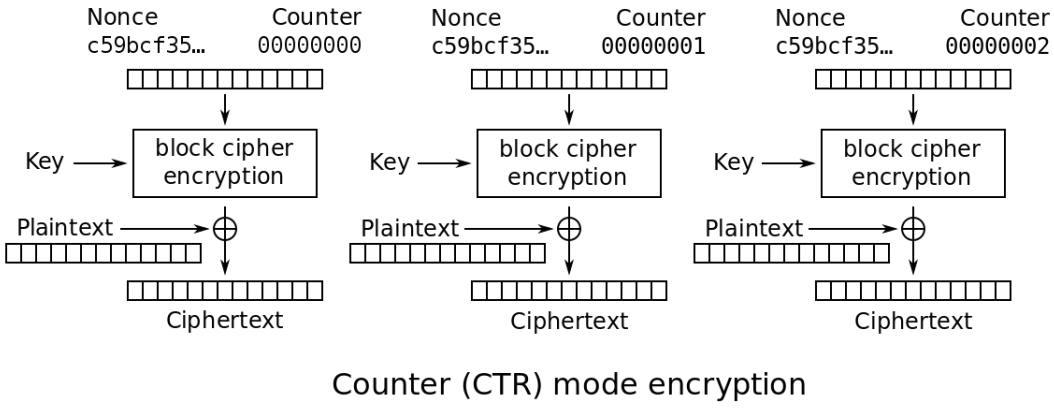


Figure 10.3: AES counter mode

Note that the nonce in this diagram is equivalent to the initialization vector (IV) in the other diagrams. However, if the offset/location information is corrupt, it will be impossible to partially recover such data due to the dependence on byte offset.

You can see details of counter mode in figure 10.4.

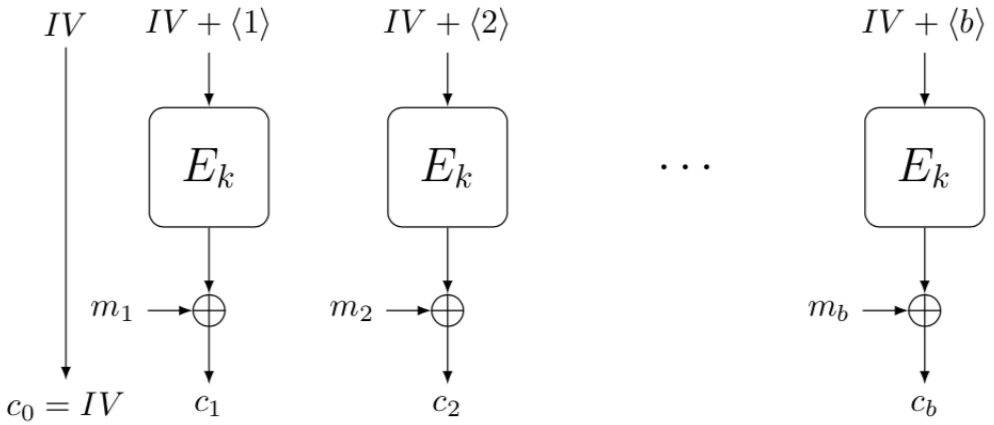


Figure 10.4: AES counter mode

At first, should choose an IV randomly from $\{0,1\}^n$ then the purpose is finding every c in this way:

$$c_i = m_i \oplus E(IV + < i >),\ i = 1,2,...,b$$

In the above equation $< i >$ will illustrate binary format for number i, also $IV + < i >$ is modular summation with the base of 2^n.
One of the advantages for CTR operation mode is that we can parallelize the encryption and decryption procedure in this mode.

Implementation in OpenSSL

OpenSSL public library provide us two groups of API for AES. One of them is EVP, and the other one is AES low level functions.

EVP api

EVP is the high level API for several kind of cryptography methods, like ECDH, RSA, HMAC .... With EVP we can make every kind of Keys and store them to relevant structures. Here we want to explain some functions in EVP library:

- EVP_CIPHER_CTX_new:
  In this function only we want to allocate storage for our cipher context object. Before any action we should do this.
- EVP_aes_128_ctr:
  It is an architecture dependent function that makes new EVP_CIPHER for AES in counter mode.
- EVP_EncryptInit:
  Using this function you can initialize our new cipher context with new IV and cipher type.
- EVP_EncryptUpdate:
  Finally using this function we can encrypt plain text in the way that we want. In AES counter mode there is no worrying about plain text length; thus it's truly easy to use.

You can find a simple example with EVP for AES in counter mode in this link

`https://github.com/RadNi/Tor-Book/code/aes_ctr_test.c`

10.3 RSA

10.3.1 MGF

A mask generation function (MGF) is a cryptographic primitive similar to a cryptographic hash function except that while a hash function's output is a fixed size, a MGF supports output of a variable length. There may be restrictions on the length of the input and output octet strings, but such bounds are generally very largeMask generation functions are completely deterministic: for any given input and desired output length the output is always the same.

MGF1

MGF1 is a Mask Generation Function based on a hash function.

Algorithm overview

MGF1 (*mgfSeed*, *maskLen*):

Options:	Hash	function (*hLen* denotes the length in octets of the hash function output)
Input:	*mgfSeed*	seed from which mask is generated, an octet string
	maskLen	intended length in octets of the mask, at most 2^{32} *hLen*
Output:	*mask*	mask, an octet string of length *maskLen*
Error:	"mask too long"	

Steps:

1. If $maskLen > 2^{32}\ hLen$, output "mask too long" and stop.
2. Let T be the empty octet string.
3. For *counter* from 0 to $\lceil maskLen/hLen \rceil$-1 , do the following:
   A. Convert counter to an octet string C of length 4 octets:

      C = I2OSP[5] (*counter*, 4).

   B. Concatenate the hash of the seed *mgfSeed* and C to the octet string T:

      $T = T||Hash(mgfSeed||C)$.

4. Output the leading *maskLen* octets of T as the octet string *mask*.

It is an example python code for MGF1.

[5] You can find detail of the I2OSP function [7].

Listing 10.1: MGF1 example on python

```
1  import hashlib
2
3  def i2osp(integer, size=4):
4    return ''.join([chr((integer >> (8 * i)) & 0xFF) for i in
5      reversed(range(size))])
6
7  def mgf1(input, length, hash=hashlib.sha1):
8    counter = 0
9    output = ''
10   while (len(output) < length):
11     C = i2osp(counter, 4)
12     output += hash(input + C).digest()
13     counter += 1
14   return output[:length]
```

10.3.2 OAEP

In cryptography, Optimal Asymmetric Encryption Padding [6] is a padding scheme often used together with RSA encryption. This processing is proved to result in a combined scheme which is semantically secure under chosen plaintext attack [7].

Encryption operation

RSAES-OAEP-ENCRYPT $((n, e), M, L)$:

Options:	Hash	hash function (*hLen* denotes the length in octets of the hash function output)
	MGF	mask generation function
Input:	(n, e)	recipient's RSA public key (k denotes the length in octets of the RSA modulus n)
	M	message to be encrypted, an octet string of length *mLen*, where $mLen \leq k - 2hLen - 2$
	L	optional label to be associated with the message; the default value for L, if L is not provided, is the empty string
Output:	C	ciphertext, an octet string of length k
Error:		"message too long"; "label too long"
Assumption:		RSA public key (n, e) is valid

[6] OAEP
[7] IND-CPA

Steps:

1. *Lengthchecking*:
   a. If the length of L is greater than the input limitation for the hash function ($2^{61}-1$ octets for SHA-1), output "label too long" and stop.

   b. If $mLen > k - 2hLen - 2$, output "message too long" and stop.

2. $EME-OAEPencoding$ (see Figure 10.5 below):
   a. If the label L is not provided, let L be the empty string. Let $lHash$ = Hash (L), an octet string of length $hLen$.

   b. Generate an octet string PS consisting of $k - mLen - hLen - 2$ zero octets. The length of PS may be zero.

   c. Concatenate $lHash$, PS, a single octet with hexadecimal value $0x01$, and the message M to form a data block DB of length $k - hLen - 1$ octets as:

   $$DB = lHash \,||\, PS \,||\, 0x01 \,||\, M.$$

   d. Generate a random octet string $seed$ of length $hLen$.

   e. Let $dbMask$ = MGF ($seed$, $k - hLen - 1$)

   f. Let $maskedDB$ = $DB \oplus dbMask$.

   g. Let $seedMask$ = MGF ($maskedDB$, $hLen$).

   h. Let maskedSeed = $seed \oplus seedMask$.

   i. Concatenate a single octet with hexadecimal value $0x00$, maskedSeed, and $maskedDB$ to form an encoded message EM of length k octets as:

   $$EM = 0x00 \,||\, maskedSeed \,||\, maskedDB.$$

3. *RSAencryption*:
   a. Convert the encoded message EM to an integer message representative m:

   $$m = \text{OS2IP}(EM)\ .$$

   b. Apply the RSAEP encryption primitive[8] to the RSA public key (n, e) and the message representative m to produce an integer ciphertext representative c:

   $$c = \text{RSAEP}((n,e),m)$$

   c. Convert the ciphertext representative c to a ciphertext C of length k octets:

   $$C = \text{I2OSP}(c,k).$$

[8] Section 5.1.1 of [7]

4. Output the ciphertext C.

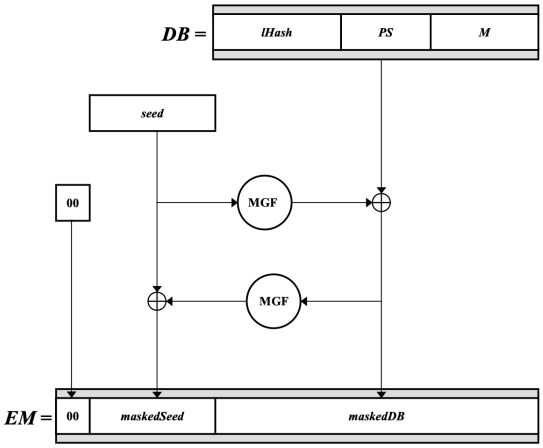


Figure 10.5: $lHash$ is the hash of the optional label L. Decoding operation follows reverse steps to recover M and verify $lHash$ and PS.

Note. If L is the empty string, the corresponding hash value $lHash$ has the following hexadecimal representation for SHA-1 Hash:

(0x)da39a3ee 5e6b4b0d 3255bfef 95601890 afd80709

10.4 SHA-1

In cryptography, SHA-1 (Secure Hash Algorithm 1) is a cryptographic hash function which takes an input and produces a 160-bit (20 byte) hash value known as a message digest

10.4.1 Algorithm overview

In figure you can see one iteration within the SHA-1 compression function: [9]

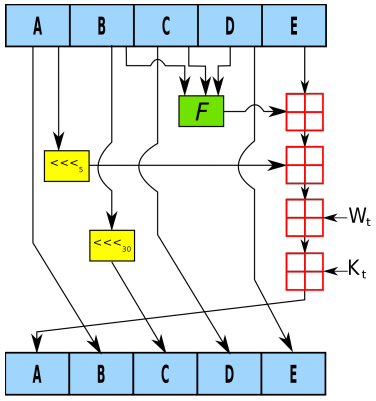


Figure 10.6: One iteration of SHA-1. A, B, C, D and E are 32-bit words of the state; F is a nonlinear function that varies; W_t is the expanded message word of round t; K_t is the round constant of round t;

[9] You can find more details of the figure here [8].

Steps[1]:

1. Padding of bits.
2. Append length.
3. Devide the input into 512-bit blocks.

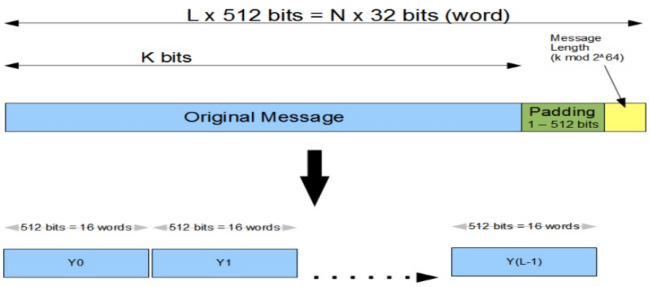


Figure 10.7: Steps 1 to 3

4. Initializing chaining variables

Chaining Variables	Hex values
A	01 23 45 67
B	89 AB CD EF
C	FE DC BA 98
D	76 54 32 10
E	C3 D2 E1 F0

Figure 10.8: Steps 4

5. Process blocks
   a. Copy chaining variables A-E into variables a-e.
   b. Divide current 512-bit block into 16 sub-blocks of 32-bit.
   c. SHA has 4 round, each consisting of 20 steps.
   Each round takes 3 inputs:
   512-bit block.
   The register abcde.
   A constant K[t] (where t = 0 to 79)

Round	Value of t between
1	1 and 19
2	20 and 39
3	40 and 59
4	60 and 79

Figure 10.9: Steps 4

   d. SHA has a total of 80 iterations (4 round X 20 iterations). Each iterations consist of the following operations:

$$abcde = (e + Processe + S^5(a) + W[t] + K[t]) \;||\; a \;||\; S^{30}(b) \;||\; c \;||\; d$$

abcde The register made up of 5 variables a, b, c, d, e.
Process P: A logical operation.10.10
S^t is a circular-shift of 32-bit sub-block by t bits.
W[t] is a 32-bit derived from the current 32-bit sub-block.
K[t] is one of the 5 additive constants.

Round	Process P
1	(b AND c) OR ((NOT b) AND (d))
2	b XOR c XOR d
3	(b AND c) OR (b AND d) OR (c AND d)
4	b XOR c XOR d

Figure 10.10: Steps 4

The values of W[t] are calculated as follows:

- For the first 16 words of W (t = 0 to t = 15), the contents of the input message sub-block M[t] become the contents of W[t].
- For the remaining 64 values of W are derived using the equation:
  $W[t] = S^1(W[t-16] \oplus W[t-14] \oplus W[t-8] \oplus W[t-3])$

Bibliography

Articles

[4] Syverson, Paul Dingledine, Roger Mathewson, Nick. “Tor: The Second-Generation Onion Router”. In: (2004) (cited on pages 11, 58, 61).

Books

URLs

[1] Vishakha Agarwal. *Secure Hash Algorithm*. https://www.slideshare.net/VishakhaAgarwal4/secure-hash-algorithm. [Online; accessed 22-Feb-2019]. 2015 (cited on page 73).

[2] *Are TOR Hidden Services really hidden? Demystifying HS Directory surveillance by injecting Decoys inside TOR!* https://www.slideshare.net/Abhinav_Biswas/are-tor-hidden-services-really-hidden-demystifying-hs-directory-surveillance-by-injecting-decoys-inside-tor. Accessed: 2018-04-21 (cited on pages 41, 61–63).

[3] *Cipher suit*. https://en.wikipedia.org/wiki/Cipher_suite. [Online; accessed 27-Feb-2019]. 2019 (cited on pages 13, 14).

[5] *Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*. https://tools.ietf.org/html/rfc5280. [Online; accessed 27-Feb-2019]. 2008 (cited on pages 15, 16).

[6] *.onion Domains DNS lookup*. https://tor.stackexchange.com/questions/711/onion-domains-dns-lookup. Accessed: 2018-04-21 (cited on page 61).

[7] *PKCS 1 v2.1: RSA Cryptography Standard*. ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-1/pkcs-1v2-1.pdf. [Online; accessed 22-Feb-2019]. 2002 (cited on pages 69, 71).

[8] *SHA-1*. https://en.wikipedia.org/wiki/SHA-1. [Online; accessed 22-Feb-2019]. 2019 (cited on page 72).

[9] *TLS/SSL Protocol*. https://medium.com/@iphelix/tls-ssl-protocol-84a1ecef54bd. [Online; accessed 27-Feb-2019]. 2019 (cited on page 14).