The Noise Protocol Framework

Trevor Perrin (noise@trevp.net)

Revision 25, 2016-04-04

Contents

1. Introduction

Noise is a framework for crypto protocols based on Diffie-Hellman key agreement. Noise can describe protocols that consist of a single message as well as interactive protocols.

2. Overview

2.1. Terminology

A Noise protocol begins with two parties exchanging **handshake messages**. During this **handshake phase** the parties exchange DH public keys and perform a sequence of DH operations, hashing the DH results into a shared secret key. After the handshake phase each party can use this shared key to send encrypted **transport messages**.

The Noise framework supports handshakes where each party has a long-term **static key pair** and/or an **ephemeral key pair**. A Noise handshake is described by a simple language. This language consists of **tokens** which are arranged into **message patterns**. Message patterns are arranged into **handshake patterns**.

A **message pattern** is a sequence of tokens that specifies the DH public keys that comprise a handshake message, and the DH operations that are performed when sending or receiving that message. A **handshake pattern** specifies the sequential exchange of messages that comprise a handshake.

A handshake pattern can be instantiated by **DH functions**, **cipher functions**, and a **hash function** to give a concrete **Noise protocol**. An application using a Noise protocol must handle some **application responsibilities** on its own, such as indicating message lengths.

2.2. Handshake state machine

The core of Noise is a set of variables maintained by each party during a handshake, and rules for sending and receiving handshake messages by sequentially processing the tokens from a message pattern.

Each party maintains the following variables:

- **`s, e`**: The local party's static and ephemeral key pairs (which may be empty).
- **`rs, re`**: The remote party's static and ephemeral public keys (which may be empty).
- **`h`**: A value that hashes all the handshake data that's been sent and received.
- **`ck`**: A "chaining key" that hashes all previous DH outputs. Once the handshake completes, the chaining key will be used to derive the encryption keys for transport messages.
- **`k, n`**: An encryption key `k` (which may be empty) and a counter-based nonce `n`. Whenever a new DH output causes a new `ck` to be calculated, a new `k` is also calculated from the same inputs. The key `k` and nonce `n` are used to encrypt static public keys and handshake payloads, incrementing `n` with each encryption. Encryption with `k` uses an "AEAD" cipher mode and includes the current `h` value as "associated data" which is covered by the AEAD authentication tag. Encryption of static public keys and payloads provides some confidentiality and key confirmation during the handshake phase.

To send a handshake message, the sender sequentially processes each token from a message pattern. The possible tokens are:

- **`"e"`**: The sender generates a new ephemeral key pair and stores it in the `e` variable, writes the ephemeral public key as cleartext into the message buffer, and hashes the public key along with the old `h` to derive a new `h`.
- **`"s"`**: The sender writes its static public key from the `s` variable into the message buffer, encrypting it if `k` is non-empty, and hashes the output along with the old `h` to derive a new `h`.

- **`"dhee", "dhse", "dhes", "dhss"`**: The sender performs a DH between its corresponding local key pair (whether `s` or `e` is determined by the first letter following `"dh"`) and the remote public key (whether `rs` or `re` is determined by the second letter following `"dh"`). The result is hashed along with the old `ck` to derive a new `ck` and `k`, and `n` is set to zero.

After processing the final token in a handshake message, the sender then writes the payload (which may be zero-length) into the message buffer, encrypting it if `k` is non-empty, and hashes the output along with the old `h` to derive a new `h`.

As a simple example, an unauthenticated DH handshake is described by the handshake pattern:

```
-> e
<- e, dhee
```

The initiator sends the first message, which is simply an ephemeral public key. The responder sends back its own ephemeral public key. Then a DH is performed and the output is hashed into `ck`, which is the final shared key from the handshake. Note that a cleartext payload is sent in the first handshake message, and an encrypted payload is sent in the response handshake message. The application may send whatever payloads it wants.

The responder can send its static public key (under encryption) and authenticate itself via a slightly different pattern:

```
-> e
<- e, dhee, s, dhse
```

In this case, the final `ck` and `k` values are a hash of both DH results. Since the `dhse` token indicates a DH between the initiator's ephemeral key and the responder's static key, successful decryption by the initiator of the second message's payload serves to authenticate the responder to the initiator.

Note that the second message's payload may contain a zero-length plaintext, but the payload ciphertext will still contain an authentication tag, since encryption is with an AEAD mode. The second message's payload can also be used to deliver certificates for the responder's static public key.

The initiator can send *its* static public key (under encryption), and authenticate itself, using a handshake pattern with one additional message:

```
-> e
<- e, dhee, s, dhse
-> s, dhse
```

The following sections flesh out the details, and add some complications (such as pre-shared symmetric keys, and "pre-messages" that represent knowledge of the other party's public keys before the handshake). However, the core of Noise is this simple system of variables, tokens, and processing rules, which allow concise expression of a range of protocols.

3. Message format

All Noise messages are less than or equal to 65535 bytes in length. Restricting message size has several advantages:

- Simpler testing, since it's easy to test the maximum sizes.
- Reduces the likelihood of errors in memory handling, or integer overflow.
- Enables support for streaming decryption and random-access decryption of large data streams.
- Enables higher-level protocols that encapsulate Noise messages to use an efficient standard length field of 16 bits.

All Noise messages can be processed without parsing, since there are no type or length fields. Of course, Noise messages might be encapsulated within a higher-level protocol that contains type and length information. Noise messages might also encapsulate payloads that require parsing of some sort, but the payloads are opaque to Noise.

A Noise **transport message** is simply an AEAD ciphertext that is less than or equal to 65535 bytes in length, and that consists of an encrypted payload plus a 16-byte authentication tag. The details depend on the AEAD cipher function, e.g. AES256-GCM, or ChaCha-Poly1305, but the 16-byte authentication tag typically occurs at the end of the ciphertext.

A Noise **handshake message** is also less than or equal to 65535 bytes. It begins with a sequence of one or more DH public keys, as determined by its message pattern. Following the public keys will be a single payload which can be used to convey certificates or other handshake data, but can also contain a zero-length plaintext.

Static public keys and payloads will be in cleartext if they occur in a handshake pattern prior to a DH operation, and will be an AEAD ciphertext if they occur after a DH operation. (If Noise is being used with pre-shared symmetric keys, this rule is different: *all* static public keys and payloads will be encrypted; see Section 7). Like transport messages, AEAD ciphertexts will expand each encrypted field (whether static public key or payload) by 16 bytes.

For an example, consider the handshake pattern:

```
-> e
<- e, dhee, s, dhse
-> s, dhse
```

The first message consists of a cleartext public key (`"e"`) followed by a cleartext payload (remember that a payload is implicit at the end of each message pattern). The second message consists of a cleartext public key (`"e"`) followed by an encrypted public key (`"s"`) followed by an encrypted payload. The third message consists of an encrypted public key (`"s"`) followed by an encrypted payload.

Assuming each payload contains a zero-length plaintext, and DH public keys are 56 bytes, the message sizes will be:

1. 56 bytes (one cleartext public key and a cleartext payload)
2. 144 bytes (two public keys, the second encrypted, and an encrypted payload)
3. 88 bytes (one encrypted public key and an encrypted payload)

If pre-shared symmetric keys are used, the first message grows in size to 72 bytes, since the first payload becomes encrypted.

4. Crypto functions

A Noise protocol is instantiated with a concrete set of **DH functions**, **cipher functions**, and a **hash function**. The signature for these functions is defined below. Some concrete functions are defined in Section 10.

Noise depends on the following **DH functions** (and an associated constant):

- **`GENERATE_KEYPAIR()`**: Generates a new DH keypair.
- **`DH(privkey, pubkey)`**: Performs a DH calculation and returns an output sequence of bytes. If the function detects an invalid public key, the output may be all zeros or any other value that doesn't leak information about the private key. For reasons discussed in Section 9.1 it is recommended for the function to have a **null public key value** that always yields the same output, regardless of private key. For example, the DH functions in Section 10 always map a DH public key of all zeros to an output of all zeros.
- **`DHLEN`** = A constant specifying the size of public keys in bytes.

Noise depends on the following **cipher functions**:

- **`ENCRYPT(k, n, ad, plaintext)`**: Encrypts `plaintext` using the cipher key `k` of 32 bytes and an 8-byte unsigned integer nonce `n` which must be unique for the key `k`. Returns the ciphertext. Encryption must be done with an "AEAD" encryption mode with the associated data `ad` and returns a ciphertext that is the same size as the plaintext plus 16 bytes for an authentication tag.
- **`DECRYPT(k, n, ad, ciphertext)`**: Decrypts `ciphertext` using a cipher key `k` of 32 bytes, an 8-byte unsigned integer nonce `n`, and associated data `ad`. Returns the plaintext, unless authentication fails, in which case an error is signaled to the caller.

Noise depends on the following **hash function** (and associated constants):

- **`HASH(data)`**: Hashes some arbitrary-length data with a collision-resistant hash function and returns an output of `HASHLEN` bytes.
- **`HASHLEN`** = A constant specifying the size in bytes of the hash output. Must be 32 or 64.
- **`BLOCKLEN`** = A constant specifying the size in bytes that the hash function uses internally to divide its input for iterative processing. This is needed to use the hash function with HMAC (`BLOCKLEN` is `B` in RFC 2104).

Noise defines an additional function based on the above `HASH()` function:

- **`HKDF(chaining_key, input_key_material)`**: Takes a `chaining_key` byte sequence of length `HASHLEN`, and an `input_key_material` byte sequence of arbitrary length. Returns two byte sequences of length `HASHLEN`, as follows. (The `HMAC-HASH(key, data)` function applies `HMAC` from RFC 2104 using the `HASH()` function; the `||` operator concatenates byte sequences; the `byte()` function constructs a single byte):
  - Sets `temp_key = HMAC-HASH(chaining_key, input_key_material)`.
  - Sets `output1 = HMAC-HASH(temp_key, byte(0x01))`.
  - Sets `output2 = HMAC-HASH(temp_key, output1 || byte(0x02))`.
  - Returns the pair `(output1, output2)`.

Note that `temp_key`, `output1`, and `output2` are all `HASHLEN` bytes in length.

5. Processing rules for handshake and transport messages

To precisely define the processing rules we adopt an object-oriented terminology, and present three "objects" which encapsulate state variables and provide "methods" which implement processing logic. These three objects are presented as a hierarchy: each higher-layer object includes one instance of the object beneath it. From lowest-layer to highest, the objects are:

- A **`CipherState`** object contains `k` and `n` variables, which it uses to encrypt and decrypt ciphertexts. During the handshake phase each party has a single `CipherState`, but during the transport phase each party has two `CipherState` objects: one for sending, and one for receiving.
- A **`SymmetricState`** object contains a `CipherState` plus `ck` and `h` variables. It is so-named because it encapsulates all the "symmetric crypto" used by Noise. During the handshake phase each party has a single `SymmetricState`, which can be deleted once the handshake is finished.

- A `HandshakeState` object contains a `SymmetricState` plus DH variables `(s, e, rs, re)` and some variables representing the handshake pattern. During the handshake phase each party has a single `HandshakeState`, which can be deleted once the handshake is finished.

To execute a Noise protocol you `Initialize()` a `HandshakeState`. During initialization you specify the handshake pattern, any local key pairs, and any public keys for the remote party you have knowledge of. After `Initialize()` you call `WriteMessage()` and `ReadMessage()` on the `HandshakeState` to process each handshake message. If a decryption error occurs the handshake has failed and the `HandshakeState` is deleted without sending further messages.

Processing the final handshake message returns two `CipherState` objects, the first for encrypting transport messages from initiator to responder, and the second for messages in the other direction. At that point the `HandshakeState` may be deleted. Transport messages are then encrypted and decrypted by calling `EncryptWithAd()` and `DecryptWithAd()` on the relevant `CipherState` with zero-length associated data.

The below sections describe these objects in detail.

5.1 The `CipherState` object

A `CipherState` can encrypt and decrypt data based on its `k` and `n` variables:

- **`k`**: A cipher key of 32 bytes (which may be `empty`). `Empty` is a special value which indicates `k` has not yet been initialized.
- **`n`**: An 8-byte (64-bit) unsigned integer nonce.

A `CipherState` responds to the following methods. The `++` post-increment operator applied to `n` means "use the current `n` value, then increment it". If incrementing `n` causes an arithmetic overflow (i.e. would result in `n` greater than 2^64 - 1) then any further `Encrypt()` or `Decrypt()` calls will signal an error to the caller.

- **`InitializeKey(key)`**: Sets `k = key`. Sets `n = 0`.
- **`HasKey()`**: Returns true if `k` is non-empty, false otherwise.
- **`EncryptWithAd(ad, plaintext)`**: If `k` is non-empty returns `ENCRYPT(k, n++, ad, plaintext)`. Otherwise returns `plaintext`.
- **`DecryptWithAd(ad, ciphertext)`**: If `k` is non-empty returns `DECRYPT(k, n++, ad, ciphertext)`. Otherwise returns `ciphertext`. If an authentication failure occurs in `DECRYPT()` the error is signaled to the caller.

5.2. The `SymmetricState` object

A `SymmetricState` object contains a `CipherState` plus the following variables:

- **`ck`**: A chaining key of `HASHLEN` bytes.
- **`h`**: A hash output of `HASHLEN` bytes.

A `SymmetricState` responds to the following methods:

- **`InitializeSymmetric(protocol_name)`**: Takes an arbitrary-length `protocol_name` byte sequence (see Section 11). Executes the following steps:
  - If `protocol_name` is less than or equal to `HASHLEN` bytes in length, sets `h` equal to `protocol_name` with zero bytes appended to make `HASHLEN` bytes. Otherwise sets `h = HASH(protocol_name)`.
  - Sets `ck = h`.
  - Calls `InitializeKey(empty)`.
- **`MixKey(input_key_material)`**: Sets `ck, temp_k = HKDF(ck, input_key_material)`. If `HASHLEN` is 64, then `temp_k` is truncated to 32 bytes to match `k`. Calls `InitializeKey(temp_k)`.
- **`MixHash(data)`**: Sets `h = HASH(h || data)`.
- **`EncryptAndHash(plaintext)`**: Sets `ciphertext = EncryptWithAd(h, plaintext)`, calls `MixHash(ciphertext)`, and returns `ciphertext`.
- **`DecryptAndHash(ciphertext)`**: Sets `plaintext = DecryptWithAd(h, ciphertext)`, calls `MixHash(ciphertext)`, and returns `plaintext`.
- **`Split()`**: Returns a pair of `CipherState` objects for encrypting transport messages. Executes the following steps:
  - Sets `temp_k1, temp_k2 = HKDF(ck, zerolen)` where `zerolen` is a zero-length byte sequence.
  - If `HASHLEN` is 64, then truncates `temp_k1` and `temp_k2` to 32 bytes apiece to match `k`.
  - Creates two new `CipherState` objects `c1` and `c2`.
  - Calls `c1.InitializeKey(temp_k1)` and `c2.InitializeKey(temp_k2)`.
  - Returns the pair `(c1, c2)`.

5.3. The `HandshakeState` object

A `HandshakeState` object contains a `SymmetricState` plus the following variables, any of which may be `empty`. `Empty` is a special value which indicates the variable has not yet been initialized.

- `s`: The local static key pair
- `e`: The local ephemeral key pair
- `rs`: The remote party's static public key
- `re`: The remote party's ephemeral public key

A `HandshakeState` also has the following variables:

- **`message_patterns`**: A sequence of message patterns. Each message pattern is a sequence of tokens from the set `("s", "e", "dhee", "dhes", "dhse", "dhss")`.
- **`message_index`**: An integer indicating the next pattern to fetch from `message_patterns`.

A `HandshakeState` responds to the following methods:

- **`Initialize(handshake_pattern, initiator, prologue, s, e, rs, re)`**: Takes a valid handshake pattern (see Section 8) and an `initiator` boolean specifying this party's role as either initiator or responder. Takes a `prologue` byte sequence which may be zero-length, or which may contain context information that both parties want to confirm is identical (see Section 6). Takes a set of DH keypairs and public keys for initializing local variables, any of which may be empty.
  - Derives a `protocol_name` byte sequence by combining the names for the handshake pattern and crypto functions, as specified in Section 11. Calls `InitializeSymmetric(protocol_name)`.
  - Calls `MixHash(prologue)`.
  - Sets the `s`, `e`, `rs`, and `re` variables to the corresponding arguments.
  - Calls `MixHash()` once for each public key listed in the pre-messages from `handshake_pattern`, with the specified public key as input (see Section 8 for an explanation of pre-messages). If both initiator and responder have pre-messages, the initiator's public keys are hashed first.
  - Sets `message_patterns` to the message patterns from `handshake_pattern`.
  - Sets `message_index = 0` (i.e. the first message pattern).
- **`WriteMessage(payload, message_buffer)`**: Takes a `payload` byte sequence which may be zero-length, and a `message_buffer` to write the output into.
  - Fetches the next message pattern from `message_patterns[message_index]`, increments `message_index`, and sequentially processes each token from the message pattern:

    - For `"e"`: Sets `e = GENERATE_KEYPAIR()`, overwriting any previous value for `e`. Appends `e.public_key` to the buffer. Calls `MixHash(e.public_key)`.
    - For `"s"`: Appends `EncryptAndHash(s.public_key)` to the buffer.
    - For `"dhxy"`: Calls `MixKey(DH(x, ry))`.
    - Appends `EncryptAndHash(payload)` to the buffer.
    - If there are no more message patterns returns two new `CipherState` objects by calling `Split()`.

- **`ReadMessage(message, payload_buffer)`**: Takes a byte sequence containing a Noise handshake message, and a `payload_buffer` to write the message's plaintext payload into.
    - Fetches the message pattern from `message_patterns[message_index]`, increments `message_index`, and sequentially processes each token from the message pattern:
    - For `"e"`: Sets `re` to the next `DHLEN` bytes from the message. Calls `MixHash(re.public_key)`.
    - For `"s"`: Sets `temp` to the next `DHLEN + 16` bytes of the message if `HasKey() == True`, or to the next `DHLEN` bytes otherwise. Sets `rs` to `DecryptAndHash(temp)`.
    - For `"dhxy"`: Calls `MixKey(DH(y, rx))`.
    - Calls `DecryptAndHash()` on the remaining bytes of the message and stores the output into `payload_buffer`.
    - If there are no more message patterns returns two new `CipherState` objects by calling `Split()`.

6. Prologue

Noise handshakes have a **prologue** input which allows arbitrary data to be hashed into the `h` variable. If both parties do not provide identical prologue data, the handshake will fail due to a decryption error. This is useful when the parties engaged in negotiation prior to the handshake and want to ensure they share identical views of that negotiation.

For example, suppose Bob communicates to Alice a list of Noise protocols that he is willing to support. Alice will then choose and execute a single protocol. To ensure that a “man-in-the-middle” did not edit Bob's list to remove options, Alice and Bob could include the list as prologue data.

Note that while the parties confirm their prologues are identical, they don't mix prologue data into encryption keys. If an input contains secret data that's intended to strengthen the encryption, a "PSK" handshake should be used instead (see next section).

7. Pre-shared symmetric keys

Noise provides an optional **pre-shared symmetric key** or **PSK** mode to support protocols where both parties already have a shared secret key. When using pre-shared symmetric keys, the following changes are made:

- Protocol names (Section 11) use the prefix `"NoisePSK_"` instead of `"Noise_"`.
- `Initialize()` takes an additional `psk` argument, which is a sequence of bytes. Immediately after `MixHash(prologue)` it sets `ck, temp = HKDF(ck,     psk)`, then calls `MixHash(temp)`. This mixes the pre-shared key into the chaining key, and also mixes a one-way function of the pre-shared key into the `h` value to ensure that `h` is a function of all handshake inputs.
- `WriteMessage()` and `ReadMessage()` are modified when processing the `"e"` token to call `MixKey(e.public_key)` as the final step. Because the initial messages in a handshake pattern are required to start with `"e"` (Section 8.1), this ensures `k` is initialized from the pre-shared key. This also uses the ephemeral public key's value as a random nonce to prevent re-using the same `k` and `n` for different messages.

8. Handshake patterns

A **message pattern** is some sequence of tokens from the set `("e", "s", "dhee", "dhes", "dhse", "dhss")`. A **handshake pattern** consists of:

- A pattern for the initiator's **pre-message** that is either:
  - `"s"`
  - `"e"`
  - `"s, e"`
  - empty
- A pattern for the responder's pre-message that takes the same range of values as the initiator's pre-message.
- A sequence of message patterns for the actual handshake messages

The pre-messages represent an exchange of public keys that was somehow performed prior to the handshake, so these public keys must be inputs to `Initialize()` for the "recipient" of the pre-message.

The first actual handshake message is sent from the initiator to the responder, the next is sent by the responder, the next from the initiator, and so on in alternating fashion.

The following handshake pattern describes an unauthenticated DH handshake:

```
Noise_NN():
  -> e
  <- e, dhee
```

The handshake pattern name is `Noise_NN`. This naming convention will be explained in Section 8.3. The empty parentheses indicate that neither party is initialized with any key pairs. The tokens `"s"` and/or `"e"` inside the parentheses would indicate that the initiator is initialized with static and/or ephemeral key pairs. The tokens `"rs"` and/or `"re"` would indicate the same thing for the responder.

Pre-messages are shown as patterns prior to the delimiter "...", with a right-pointing arrow for the initiator's pre-message, and a left-pointing arrow for the responder's pre-message. If both parties have a pre-message, the initiator's is listed first (and hashed first). During `Initialize()`, `MixHash()` is called on any pre-message public keys, as described in Section 5.3.

The following pattern describes a handshake where the initiator has pre-knowledge of the responder's static public key, and performs a DH with the responder's static public key as well as the responder's ephemeral public key. This pre-knowledge allows an encrypted payload to be sent in the first message, although full forward secrecy and replay protection is only achieved with the second message.

```
Noise_NK(rs):
  <- s
  ...
  -> e, dhes
  <- e, dhee
```

8.1 Pattern validity

Noise patterns must be **valid** in the following senses:

- Parties can only send a static public key if they were initialized with a static key pair, and can only perform DH between private keys and public keys they possess.

- Parties must send a fresh ephemeral public key at the start of the first message they send (i.e. the first token of the first message pattern in each direction must be `"e"`).
- After performing a DH between a remote public key and any local private key that is not a "fresh" ephemeral private key, the local party must not send any payloads or static public keys, nor complete the handshake, unless they have also performed a DH between a "fresh" ephemeral private key and the remote public key. A "fresh" ephemeral private key is one that was created by processing an `"e"` token when sending a message (as opposed to an ephemeral private key passed in during initialization).

Patterns failing the first check will obviously abort the program.

The second and third checks are necessary because Noise uses DH outputs involving ephemeral keys to randomize the shared secret keys. Noise also uses ephemeral public keys to randomize PSK-based encryption. Patterns failing these checks could result in subtle but catastrophic security flaws.

Users are recommended to only use the handshake patterns listed below, or other patterns that have been vetted by experts to satisfy the above checks.

8.2. One-way patterns

The following example handshake patterns represent "one-way" handshakes supporting a one-way stream of data from a sender to a recipient. These patterns could be used to encrypt files, database records, or other non-interactive data streams.

Following a one-way handshake the sender can send a stream of transport messages, encrypting them using the first `CipherState` returned by `Split()`. The second `CipherState` from `Split()` is discarded - the recipient must not send any messages using it (as this would violate the rules in Section 8.1).

```
Naming convention for one-way patterns:
  N = no static key for sender
  K = static key for sender known to recipient
  X = static key for sender transmitted to recipient

Noise_N(rs):
  <- s
  ...
  -> e, dhes

Noise_K(s, rs):
  -> s
  <- s
  ...
  -> e, dhes, dhss

Noise_X(s, rs):
  <- s
  ...
  -> e, dhes, s, dhss
```

`Noise_N` is a conventional DH-based public-key encryption. The other patterns add sender authentication, where the sender's public key is either known to the recipient beforehand (`Noise_K`) or transmitted under encryption (`Noise_X`).

8.3. Interactive patterns

The following example handshake patterns represent interactive protocols.

```
Naming convention for interactive patterns:

  N_ = no static key for initiator
  K_ = static key for initiator known to responder
  X_ = static key for initiator transmitted to responder
  I_ = static key for initiator immediately transmitted to responder,
       despite reduced or absent identity-hiding

  _N = no static key for responder
  _K = static key for responder known to initiator
  _X = static key for responder transmitted to initiator
  _R = static key for responder transmitted to initiator, but only
       after the initiator has revealed their identity


Noise_NN():                       Noise_KN(s):
  -> e                              -> s
  <- e, dhee                        ...
                                    -> e
                                    <- e, dhee, dhes

Noise_NK(rs):                     Noise_KK(s, rs):
  <- s                              -> s
  ...                               <- s
  -> e, dhes                        ...
  <- e, dhee                        -> e, dhes, dhss
                                    <- e, dhee, dhes

Noise_NX(rs):                     Noise_KX(s, rs):
  -> e                              -> s
  <- e, dhee, s, dhse               ...
```

```
                                  -> e
                                  <- e, dhee, dhes, s, dhse


Noise_XN(s):                    Noise_IN(s):
  -> e                            -> e, s
  <- e, dhee                      <- e, dhee, dhes
  -> s, dhse

Noise_XK(s, rs):                Noise_IK(s, rs):
  <- s                            <- s
  ...                             ...
  -> e, dhes                      -> e, dhes, s, dhss
  <- e, dhee                      <- e, dhee, dhes
  -> s, dhse

Noise_XX(s, rs):                Noise_IX(s, rs):
  -> e                            -> e, s
  <- e, dhee, s, dhse             <- e, dhee, dhes, s, dhse
  -> s, dhse

Noise_XR(s, rs):
  -> e
  <- e, dhee
  -> s, dhse
  <- s, dhse
```

The `Noise_XX` pattern is the most generically useful, since it is efficient and supports mutual authentication and transmission of static public keys. The `Noise_XR` pattern is similar to `Noise_XX` but is less efficient, and offers stronger identity-hiding for the responder rather than the initiator. (This is similar to the distinction between SIGMA-I and SIGMA-R; see Section 8.5 for more analysis on identity-hiding.)

All interactive patterns allow some encryption of handshake payloads:

- Patterns where the initiator has pre-knowledge of the responder's static public key (i.e. patterns ending in `"K"`) allow "zero-RTT" encryption, meaning the initiator can encrypt the first handshake payload.
- All interactive patterns allow "half-RTT" encryption of the first response payload, but the encryption only targets an initiator static public key in patterns starting with "K" or "I".

The security properties for handshake payloads are usually weaker than the final security properties achieved by transport payloads, so these early encryptions must be used with caution.

In some patterns the security properties of transport payloads can also vary. In particular: patterns starting with “K” or “I” have the caveat that the responder is only guaranteed “weak” forward secrecy for the transport messages it sends until it receives a transport message from the initiator. After receiving a transport message from the initiator, the responder becomes assured of “strong” forward secrecy.

The next section provides more analysis of these payload security properties.

8.4. Payload security properties

The following table lists the security properties for Noise handshake and transport payloads for all the named patterns in Section 8.2 and Section 8.3. Each payload is assigned an “authentication” property regarding the degree of authentication of the sender provided to the recipient, and a “confidentiality” property regarding the degree of confidentiality provided to the sender.

The authentication properties are:

- **0 = No authentication.** This payload may have been sent by any party, including an active attacker.
- **1 = Sender authentication *vulnerable* to key-compromise impersonation (KCI)**. The sender authentication is based on a static-static DH (`"dhss"`) involving both parties’ static key pairs. If the recipient’s long-term private key has been compromised, this authentication can be forged. Note that a future version of Noise might include signatures, which could improve this security property, but brings other trade-offs.
- **2 = Sender authentication *resistant* to key-compromise impersonation (KCI)**. The sender authentication is based on an ephemeral-static DH (`"dhes"` or `"dhse"`) between the sender’s static key pair and the recipient’s ephemeral key pair. Assuming the corresponding private keys are secure, this authentication cannot be forged.

The confidentiality properties are:

- **0 = No confidentiality.** This payload is sent in cleartext.
- **1 = Encryption to an ephemeral recipient.** This payload has forward secrecy, since encryption involves an ephemeral-ephemeral DH (`"dhee"`). However, the sender has not authenticated the recipient, so this payload might be sent to any party, including an active attacker.
- **2 = Encryption to a known recipient, forward secrecy for sender compromise only, vulnerable to replay.** This payload is encrypted based only on DHs involving the recipient’s static key pair. If the recipient’s static private key is compromised, even at a later date, this payload can be decrypted. This message can also be replayed, since there’s no ephemeral contribution from the recipient.

- **3 = Encryption to a known recipient, weak forward secrecy.** This payload is encrypted based on an ephemeral-ephemeral DH and also an ephemeral-static DH involving the recipient's static key pair. However, the binding between the recipient's alleged ephemeral public key and the recipient's static public key hasn't been verified by the sender, so the recipient's alleged ephemeral public key may have been forged by an active attacker. In this case, the attacker could later compromise the recipient's static private key to decrypt the payload. Note that a future version of Noise might include signatures, which could improve this security property, but brings other trade-offs.
- **4 = Encryption to a known recipient, weak forward secrecy if the sender's private key has been compromised.** This payload is encrypted based on an ephemeral-ephemeral DH, and also based on an ephemeral-static DH involving the recipient's static key pair. However, the binding between the recipient's alleged ephemeral public and the recipient's static public key has only been verified based on DHs involving both those public keys and the sender's static private key. Thus, if the sender's static private key was previously compromised, the recipient's alleged ephemeral public key may have been forged by an active attacker. In this case, the attacker could later compromise the intended recipient's static private key to decrypt the payload (this is a variant of a "KCI" attack enabling a "weak forward secrecy" attack). Note that a future version of Noise might include signatures, which could improve this security property, but brings other trade-offs.
- **5 = Encryption to a known recipient, strong forward secrecy.** This payload is encrypted based on an ephemeral-ephemeral DH and well as an ephemeral-static DH with the recipient's static key pair. Assuming the ephemeral private keys are secure, and the recipient is not being actively impersonated by an attacker that has stolen its static private key, this payload cannot be decrypted.

For one-way handshakes, the below-listed security properties apply to the handshake payload as well as transport payloads.

For interactive handshakes, security properties are listed for each handshake payload. Transport payloads are listed as arrows without a pattern. Transport payloads are only listed if they have different security properties than the previous handshake payload sent from the same party. If two transport payloads are listed, the security properties for the second only apply if the first was received.

	Authentication	Confidentiality
Noise_N	0	2
Noise_K	1	2
Noise_X	1	2

```
Noise_NN
  -> e                          0                0
  <- e, dhee                    0                1
  ->                            0                1

Noise_NK
  <- s
  ...
  -> e, dhes                    0                2
  <- e, dhee                    2                1
  ->                            0                5

Noise_NX
  -> e                          0                0
  <- e, dhee, s, dhse           2                1
  ->                            0                5


Noise_XN
  -> e                          0                0
  <- e, dhee                    0                1
  -> s, dhse                    2                1
  <-                            0                5

Noise_XK
  <- s
  ...
  -> e, dhes                    0                2
  <- e, dhee                    2                1
  -> s, dhse                    2                5
  <-                            2                5

Noise_XX
 -> e                           0                0
 <- e, dhee, s, dhse            2                1
 -> s, dhse                     2                5
 <-                             2                5

Noise_XR
  -> e                          0                0
  <- e, dhee                    0                1
  -> s, dhse                    2                1
  <- s, dhse                    2                5
  ->                            2                5
```

```
Noise_KN
  -> s
  ...
  -> e                          0                 0
  <- e, dhee, dhes              0                 3
  ->                            2                 1
  <-                            0                 5

Noise_KK
  -> s
  <- s
  ...
  -> e, dhes, dhss              1                 2
  <- e, dhee, dhes              2                 4
  ->                            2                 5
  <-                            2                 5

Noise_KX
  -> s
  ...
  -> e                          0                 0
  <- e, dhee, dhes, s, dhse     2                 3
  ->                            2                 5
  <-                            2                 5


Noise_IN
  -> e, s                       0                 0
  <- e, dhee, dhes              0                 3
  ->                            2                 1
  <-                            0                 5

Noise_IK
  <- s
  ...
  -> e, dhes, s, dhss           1                 2
  <- e, dhee, dhes              2                 4
  ->                            2                 5
  <-                            2                 5

Noise_IX
  -> e, s                       0                 0
  <- e, dhee, dhes, s, dhse     2                 3
  ->                            2                 5
  <-                            2                 5
```

8.5. Identity hiding

The following table lists the identity hiding properties for all the named patterns in Section 8.2 and Section 8.3. Each pattern is assigned properties describing the confidentiality supplied to the initiator's static public key, and to the responder's static public key. The underlying assumptions are that ephemeral private keys are secure, and that parties abort the handshake if they receive a static public key from the other party which they don't trust.

This section only considers identity leakage through static public key fields in handshakes. Of course, the identities of Noise participants might be exposed through other means, including payload fields, traffic analysis, or metadata such as IP addresses.

The properties for the relevant public key are:

- **0.** Transmitted in clear.
- **1.** Encrypted with forward secrecy, but can be probed by an anonymous initiator.
- **2.** Encrypted with forward secrecy, but sent to an anonymous responder.
- **3.** Not transmitted, but a passive attacker can check candidates for the responder's private key and determine whether the candidate is correct.
- **4.** Encrypted to responder's static public key, without forward secrecy. If an attacker learns the responder's private key they can decrypt the initiator's public key.
- **5.** Not transmitted, but a passive attacker can check candidates for the responder's private key and initiator's public key and determine if both candidates are correct.
- **6.** Encrypted but with weak forward secrecy. An active attacker who pretends to be the initiator without the initiator's static private key, then later learns the initiator private key, can then decrypt the responder's public key.
- **7.** Not transmitted, but an active attacker who pretends to be the initator without the initiator's static private key, then later learns a candidate for the initiator private key, can then check whether the candidate is correct.
- **8.** Encrypted with forward secrecy to an authenticated party.

Identity hiding properties for static public keys:

	Initiator	Responder
Noise_N	-	3
Noise_K	5	5
Noise_X	4	3

Noise_NN	-	-
Noise_NK	-	3
Noise_NX	-	1
Noise_XN	2	-
Noise_XK	8	3
Noise_XX	8	1
Noise_XR	2	8
Noise_KN	7	-
Noise_KK	5	5
Noise_KX	7	6
Noise_IN	0	-
Noise_IK	4	3
Noise_IX	0	6

8.6. More patterns

The patterns in the previous sections are useful examples which we are naming for convenience. Other valid patterns could be constructed, for example:

- It would be easy to modify `Noise_X` or `Noise_IK` to transmit the sender's static public key in cleartext instead of encrypted by changing `"e, dhes, s, dhss"` to `"e, s, dhes, dhss"`. This would worsen identity hiding, so we're not bothering to name those patterns.
- It would be easy to make `Noise_KK` or `Noise_IK` slightly more efficient by removing the `"dhss"`. This would worsen security properties for the initial exchange of handshake payloads, so we're not bothering to name those patterns.
- In some patterns both initiator and responder have a static key pair, but `"dhss"` is not performed. This DH operation could be added to provide more resilience in case the ephemerals are generated by a bad RNG.
- Pre-knowledge of ephemeral public keys is not demonstrated in any of the above patterns. This pre-knowledge could be used to implement a "pre-key" or "semi-ephemeral key" handshake, where the forward-secrecy and KCI-resistance of zero-RTT data is improved since the pre-distributed "semi-ephemeral" key can be changed more frequently than the responder's static key. These patterns might benefit from signatures, which are not yet included in Noise. These patterns also introduce new complexity around the lifetimes of semi-ephemeral key pairs, so are not discussed further here.

9. Complex protocols

A handshake pattern specifies a fixed sequence of messages. In some cases parties executing a handshake pattern may discover a need to send a different sequence of messages. Noise has two ways to handle this.

9.1. Dummy static public keys

Consider a protocol where an initiator will authenticate itself if requested by the responder. This could be viewed as the initiator choosing between patterns like `Noise_NX` and `Noise_XX` based on some value inside the responder's first handshake payload.

Noise doesn't directly support this. Instead, this could be simulated by always executing `Noise_XX`. The initiator can simulate the `Noise_NX` case by sending a **dummy static public key** if authentication is not requested. The value of the dummy public key doesn't matter. For efficiency, the initiator can send a null public key value per Section 4 (e.g. an all-zeros `25519` value that is guaranteed to produce an all-zeros output).

This technique is simple, since it allows use of a single handshake pattern. It also doesn't reveal which option was chosen from message sizes. It could be extended to allow a `Noise_XX` pattern to support any permutation of authentications (initiator only, responder only, both, or none).

9.2. Handshake re-initialization and Noise Pipes

Consider a protocol where the initiator can attempt zero-RTT encryption based on the responder's static public key. If the responder has changed their static public key, the parties will need to switch to a "fallback" handshake where the responder transmits the new static public key and the initiator resends the zero-RTT data.

This can be handled by both parties re-initalizing their `HandshakeState` and simply executing a different handshake. Public keys that were exchanged in the first handshake can be represented as pre-messages in the second handshake. If any important negotiation occurred in the first handshake, the first handshake's `h` variable should be provided as prologue to the second handshake.

By way of example, this section defines the **Noise Pipe** protocol. This protocol uses two patterns defined in the previous section: `Noise_XX` is used for an initial handshake if the parties haven't communicated before, after which the initiator can cache the responder's static public key. `Noise_IK` is used for a zero-RTT handshake.

If the responder fails to decrypt the first `Noise_IK` message (perhaps due to changing her static key), the responder will initiate a new `Noise_XXfallback` handshake identical to `Noise_XX` except re-using the ephemeral public key from the first `Noise_IK` message as a pre-message public key.

Below are the three patterns used for Noise Pipes:

```
Noise_XX(s, rs):
  -> e
  <- e, dhee, s, dhse
  -> s, dhse

Noise_IK(s, rs):
  <- s
  ------
  -> e, dhes, s, dhss
  <- e, dhee, dhes

Noise_XXfallback(s, rs, re):
  <- e
  ------
  -> e, dhee, s, dhse
  <- s, dhse
```

Note that in the fallback case, the initiator and responder roles are switched: If Alice inititates a `Noise_IK` handshake with Bob, Bob might initiate a `Noise_XX_fallback` handshake.

There needs to be some way for the recipient of a message to distinguish whether it's the next message in the current handshake pattern, or requires re-initialization for a new pattern. For example, each handshake message could be preceded by a `type` byte (see Section 12). This byte is not part of the Noise message proper, but simply signals when re-initialization is needed:

- If `type == 0` in the initiator's first message then the initiator is performing a `Noise_XX` handshake.
- If `type == 1` in the initiator's first message then the initiator is performing a `Noise_IK` handshake.
- If `type == 1` in the responder's first `Noise_IK` response then the responder failed to authenticate the initiator's `Noise_IK` message and is performing a `Noise_XXfallback` handshake, using the initiator's ephemeral public key as a pre-message.
- In all other cases, `type` will be 0.

10. DH functions, cipher functions, and hash functions

10.1. The `25519` DH functions

- **`GENERATE_KEYPAIR()`**: Returns a new Curve25519 keypair.
- **`DH(privkey, pubkey)`**: Executes the Curve25519 DH function (aka "X25519" in RFC 7748). The null public key value is all zeros, which will always produce an output of all zeros. Other invalid public key values will also produce an output of all zeros.
- **`DHLEN`** = 32

10.2. The `448` DH functions

- **`GENERATE_KEYPAIR()`**: Returns a new Curve448 keypair.
- **`DH(privkey, pubkey)`**: Executes the Curve448 DH function (aka "X448" in RFC 7748). The null public key value is all zeros, which will always produce an output of all zeros. Other invalid public key values will also produce an output of all zeros.
- **`DHLEN`** = 56

10.3. The `ChaChaPoly` cipher functions

- **`ENCRYPT(k, n, ad, plaintext) / DECRYPT(k, n, ad, ciphertext)`**: `AEAD_CHACHA20_POLY1305` from RFC 7539. The 96-bit nonce is formed by encoding 32 bits of zeros followed by little-endian encoding of `n`. (Earlier implementations of ChaCha20 used a 64-bit nonce, in which case it's compatible to encode `n` directly into the ChaCha20 nonce without the 32-bit zero prefix).

10.4. The `AESGCM` cipher functions

- **`ENCRYPT(k, n, ad, plaintext) / DECRYPT(k, n, ad, ciphertext)`**: AES256-GCM from NIST SP 800-38D with 128-bit tags. The 96-bit nonce is formed by encoding 32 bits of zeros followed by big-endian encoding of `n`.

10.5. The `SHA256` hash function

- **`HASH(input)`**: `SHA2-256(input)`
- **`HASHLEN`** = 32
- **`BLOCKLEN`** = 64

10.6. The `SHA512` hash function

- **`HASH(input)`**: `SHA2-512(input)`
- **`HASHLEN`** = 64
- **`BLOCKLEN`** = 128

10.7. The `BLAKE2s` hash function

- **`HASH(input)`**: `BLAKE2s(input)` with digest length 32.
- **`HASHLEN`** = 32
- **`BLOCKLEN`** = 64

10.8. The `BLAKE2b` hash function

- **`HASH(input)`**: `BLAKE2b(input)` with digest length 64.
- **`HASHLEN`** = 64
- **`BLOCKLEN`** = 128

11. Protocol names

To produce a **Noise protocol name** for `Initialize()` you concatenate the names for the handshake pattern, the DH functions, the cipher functions, and the hash function. For example:

- `Noise_XX_25519_AESGCM_SHA256`
- `Noise_N_25519_ChaChaPoly_BLAKE2s`
- `Noise_XXfallback_448_AESGCM_SHA512`
- `Noise_IK_448_ChaChaPoly_BLAKE2b`

If a pre-shared symmetric key is in use, then the prefix `NoisePSK_` is used instead of `Noise_`:

- `NoisePSK_XX_25519_AESGCM_SHA256`
- `NoisePSK_N_25519_ChaChaPoly_BLAKE2s`
- `NoisePSK_XXfallback_448_AESGCM_SHA512`
- `NoisePSK_IK_448_ChaChaPoly_BLAKE2b`

12. Application responsibilities

An application built on Noise must consider several issues:

- **Choosing crypto functions**: The `25519` DH functions are recommended for most uses, along with either `AESGCM_SHA256` or `ChaChaPoly_BLAKE2s`. For an extreme security margin, you could use the `448` DH functions with either `AESGCM_SHA512` or `ChaChaPoly_BLAKE2b`.
- **Extensibility**: Applications are recommended to use an extensible data format for the payloads of all messages (e.g. JSON, Protocol Buffers). This ensures that fields can be added in the future which are ignored by older implementations.
- **Padding**: Applications are recommended to use a data format for the payloads of all encrypted messages that allows padding. This allows implementations to avoid leaking information about message sizes. Using an extensible data format, per the previous bullet, will typically suffice.
- **Termination**: Applications must consider that a sequence of Noise transport messages could be truncated by an attacker. Applications should include explicit length fields or termination signals inside of transport payloads to signal the end of a stream of transport messages.
- **Length fields**: Applications must handle any framing or additional length fields for Noise messages, considering that a Noise message may be up to 65535 bytes in length. If an explicit length field is needed, applications are recommended to add a 16-bit big-endian length field prior to each message.
- **Type fields**: Applications are recommended to include a single-byte type field prior to each Noise handshake message (and prior to the length field, if one is included). Applications would reject messages with unknown type. This allows extending the handshake with handshake re-initialization or other alternative messages in the future.

13. Security considerations

This section collects various security considerations:

- **Termination**: Preventing attackers from truncating a stream of transport messages is an application responsibility. See previous section.
- **Data volumes**: The `AESGCM` cipher functions suffer a gradual reduction in security as the volume of data encrypted under a single key increases. Due to this, parties should not send more than 2^56 bytes (roughly 72 petabytes) encrypted by a single key. If sending such large volumes of data is a possibility, different cipher functions should be chosen.
- **Rollback**: If parties decide on a Noise protocol based on some previous negotiation that is not included as prologue, then a rollback attack might be possible. This is a particular risk with handshake re-initialization, and requires careful attention if a Noise handshake is preceded by communication between the parties.
- **Incrementing nonces**: Reusing a nonce value for `n` with the same key `k` for encryption would be catastrophic. Implementations must carefully follow the rules for nonces. Nonces are not allowed to wrap back to zero due to integer overflow. This means parties are not allowed to send more than 2^64 transport messages.
- **Fresh ephemerals**: Every party in a Noise protocol should send a new ephemeral public key and perform a DH with it prior to sending any encrypted data. Otherwise replay of a handshake message could trigger catastrophic key reuse. This is one rationale behind the patterns in Section 8, and the validity rules in Section 8.1. It's also the reason why one-way handshakes only allow transport messages from the sender, not the recipient.
- **Protocol names**: The protocol name used with `Initialize()` must uniquely identify the combination of handshake pattern and crypto functions for every key it's used with (whether ephemeral key pair, static key pair, or PSK). If the same secret key was reused with the same protocol name but a different set of cryptographic operations then bad interactions could occur.
- **Pre-shared symmetric keys**: Pre-shared symmetric keys should be secret values with 256 bits of entropy (or more).
- **Channel binding**: Depending on the DH functions, it might be possible for a malicious party to engage in multiple sessions that derive the same shared secret key (e.g. if setting her public keys to invalid values causes DH outputs of zero, as is the case for the `25519` and `448` DH functions). If a higher-level protocol wants a unique "channel binding" value for referring to a Noise session it should use the value of `h` after the final handshake message, not `ck`.
- **Implementation fingerprinting**: If this protocol is used in settings with anonymous parties, care should be taken that implementations behave

identically in all cases. This may require mandating exact behavior for handling of invalid DH public keys.

14. Rationale

This section collects various design rationale:

Nonces are 64 bits in length because:

- Some ciphers (e.g. Salsa20) only have 64 bit nonces.
- 64 bit nonces were used in the initial specification and implementations of ChaCha20, so Noise nonces can be used with these implementations.
- 64 bits makes it easy for the entire nonce to be treated as an integer and incremented.
- 96 bits nonces (e.g. in RFC 7539) are a confusing size where it's unclear if random nonces are acceptable.

The recommended hash function families are SHA2 and BLAKE2 because:

- SHA2 is widely available.
- SHA2 is often used alongside AES.
- BLAKE2 is similar to ChaCha20.

Hash output lengths of 256 bits are supported because:

- SHA2-256 and BLAKE2s have sufficient collision-resistance at the 128-bit security level.
- SHA2-256 and BLAKE2s require less RAM, and less calculation when processing smaller inputs (due to smaller block size), then their larger brethren (SHA2-512 and BLAKE2b).
- SHA2-256 and BLAKE2s are faster on 32-bit processors than their larger brethren.

Cipher keys are 256 bits because:

- 256 bits is a conservative length for cipher keys when considering cryptanalytic safety margins, time/memory tradeoffs, multi-key attacks, and quantum attacks.

The authentication tag is 128 bits because:

- Some algorithms (e.g. GCM) lose more security than an ideal MAC when truncated.
- Noise may be used in a wide variety of contexts, including where attackers can receive rapid feedback on whether MAC guesses are correct.
- A single fixed length is simpler than supporting variable-length tags.

The GCM security limit is 2^56 bytes because:

- This is 2^52 AES blocks (each block is 16 bytes). The limit is based on the risk of birthday collisions being used to rule out plaintext guesses. The probability an attacker could rule out a random guess on a 2^56 byte plaintext is less than 1 in 1 million (roughly (2^52 * 2^52) / 2^128).

Big-endian is preferred because:

- Any Noise length fields are likely to be handled by parsing code where big-endian "network byte order" is traditional.
- Some ciphers use big-endian internally (e.g. GCM, SHA2).
- While it's true that Curve25519, Curve448, and ChaCha20/Poly1305 use little-endian, these will likely be handled by specialized libraries, so there's not a strong argument for aligning with them.

The `MixKey()` design uses `HKDF` because:

- HKDF is a conservative and widely used design.

`MixHash()` is used instead of `MixKey()` because:

- `MixHash()` is more efficient than `MixKey()`.
- `MixHash()` avoids any IPR concerns regarding mixing identity data into session keys (see KEA+).
- `MixHash()` produces a non-secret `h` value that might be useful to higher-level protocols, e.g. for channel-binding.

15. IPR

The Noise specification (this document) is hereby placed in the public domain.

16. Acknowledgements

Noise is inspired by the NaCl and CurveCP protocols from Dan Bernstein et al., and also by HOMQV and SIGMA from Hugo Krawzcyk.

Feedback on the spec came from: Moxie Marlinspike, Jason Donenfeld, Tiffany Bennett, Jonathan Rudenberg, Stephen Touset, and Tony Arcieri.

Thanks to Tom Ritter and Karthikeyan Bhargavan for editorial feedback.

Moxie Marlinspike, Christian Winnerlein, and Hugo Krawzcyk provided feedback on key derivation.

Jeremy Clark, Thomas Ristenpart, and Joe Bonneau gave feedback on much earlier versions.